Greedy Scheduling with Feedback Control for Overloaded Real-Time Systems

Zhuo Cheng, Haitao Zhang, Yasuo Tan, and Azman Osman Lim
School of Information Science, JAIST
Nomi, Ishikawa 923-1292, Japan
{chengzhuo, zhanghaitao, ytan, aolim}@jaist.ac.jp

***Abstract*—In real-time systems, a task is required to be completed before its deadline. When workload is heavy, the system may become overloaded. Under such condition, some tasks may miss their deadlines. To deal with this overload problem, the design of scheduling algorithm is crucial. In this paper, we focus on studying on-line scheduling for overloaded real-time systems. The objective is to maximize the total number of tasks that meet their deadlines. To achieve this goal, the idea of greedy algorithm is used to propose a greedy scheduling (GS) algorithm. In each time, GS makes an optimum choice for currently known task set. As the uncertainty of new arriving tasks, GS cannot make an optimum choice for the set of overall tasks. To deal with this uncertainty, by applying feedback control, a greedy scheduling with feedback control (GSFC) is introduced. Three widely used scheduling algorithms and their corresponding deferrable scheduling (DS) methods are discussed and compared with GSFC. Simulation results reveal that GSFC can effectively improve the system performance.**

I. Introduction

Cyber-Physical Systems (CPSs) are systems with tight coupling between computing and physical environment. With their rapid developments, CPSs have enlivened many critical areas for human life such as transportation, energy, and health. In CPS, as the dynamic nature of physical processing, sensitivity to timing and concurrency become central features of system behavior [1]. These features make typical CPSs as multi-tasking real-time systems. In such a system, a task is required to be completed before a specified time instant called deadline. The execution order of tasks is set by a scheduler. Under ideal workload condition, scheduler with a proper scheduling algorithm can make all tasks meet their deadlines. However, in practical environment, system workload may vary widely. Once system workload becomes too heavy so that there does not exist a feasible scheduling algorithm can make all the tasks meet their deadlines, we say the system is *overloaded*.

When overload problem happens, it is important to minimize the degrees of system performance degradation cased by tasks missing deadlines. A system that panics and suffers a drastic fall in performance when a problem happens, is likely to contribute to this problem, rather than help solve it [2]. To achieve this target, the design of scheduling algorithms is crucial, as different scheduling algorithms will lead to different degrees of performance degradation. In this paper, we focus on studying *on-line scheduling* (scheduler has no knowledge of a task until it makes request to execute) for overloaded real-time systems with uniprocessor. Our objective is to maximize the total number of tasks that meet their deadlines. This objective is reasonable upon the application that when a missed deadline corresponds to a disgruntled customer, and the aim is to keep as many customers satisfied as possible [2].

There are mainly two contributions in this paper. (*i*) Utilize the idea of greedy algorithm to present greedy scheduling (GS) which can make an optimum choice for currently known task set. Two theorems are proposed to prove that GS meets the characteristic of greedy algorithm. Although GS is designed for the objective that maximizes the total number of tasks that meet their deadlines, the method and procedures of utilizing the idea of greedy algorithm to design scheduling algorithm can be easily extended to other objectives, and can benefit other research on the design of scheduling algorithm for overloaded real-time systems. (*ii*) Extend GS to greedy scheduling with feedback control (GSFC). With the help of feedback control, GSFC can dynamically deal with the uncertainty of new arriving tasks. This uncertainty is the biggest challenge in the design of on-line scheduling algorithm. This idea gives a feasible method to meet this challenge. Moreover, this gives a way that utilizes outcomes in the control theory to benefit the design of on-line scheduling.

II. System Model, Definition, and Preliminary

A. Notation and Assumptions

We adopt general *firm-deadline* model proposed in [3]. The "firm-deadline" means only tasks completed before their deadlines are considered valuable, and any task missing its deadline is worthless to system. The real-time system comprises a set of aperiodic real-time tasks waiting to execute. These tasks request processor to execute when they arrive in system. Each task τ_i is a 3-tuple $\tau_i = (r_i, c_i, d_i)$, where i is the index of a task, r_i is the request time instant, c_i is the required execution time, and d_i is the deadline. Symbol $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ denotes the set of tasks comprised in the system, where n is the number of tasks. Task set $\mathcal{T}$ varies with the passage of time. At system time t, $\forall \tau_i \in \mathcal{T}$ meets $r_i \leq t$. Symbol rc_i represents remaining execution time of task τ_i. Initially, it equals to c_i. After τ_i has been executed for δ ($\delta \leq c_i$) time units, $rc_i = c_i - \delta$. If $rc_i = 0$, it means τ_i has been completed. A successfully completed task τ_i feathers that it has been executed c_i time units during time interval $[r_i, d_i)$. Note that, if $rc_i > d_i - t$, task τ_i should be discarded immediately, as such task cannot be able to complete successfully.

The assumptions that apply to the system model are as follows: (*i*) The scheduler can learn of a task's attributes at the time instant when it makes request, nothing is known about a

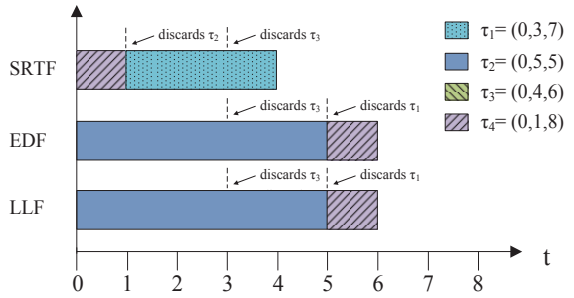

Fig. 1. Performance of scheduling algorithms

task before this time. (*ii*) A task being executed on processor can be preempted by another task at any time instant, and there is no associated cost with such preemption. (*iii*) Every task is independent with the others. There is no prior bound on the time instant and number of takes which request to execute.

B. Definition

Definition [4]. *When there exists a scheduling algorithm can make all tasks meet their deadlines, the system is* ***underloaded****, and the task set is* ***feasible****. On the contrast, when there does not exist a scheduling algorithm can make all the tasks meet their deadlines, the system is* ***overloaded****, and the task set is* ***infeasible****.*

C. Preliminary

There are many scheduling algorithms used in various real-time systems. Three representative scheduling algorithms are adopted as baseline algorithms: shortest remaining time first (SRTF), earliest deadline first (EDF), and least laxity first (LLF). We use an example described in Fig. 1 to study their performance. The scheduling results are: (*i*): SRTF first schedules task with the shortest remaining time. The scheduling sequence is $\langle \tau_4, \tau_1 \rangle$. By this sequence, τ_4 and τ_1 can be completed sequentially. (*ii*): EDF first schedules the task with the earliest deadline. It has been proven as an *optimal* scheduling algorithm on uniprocessor. That is, if using EDF to schedule a task set cannot make all tasks meet their deadlines, no other algorithms can. The result of scheduling sequence is $\langle \tau_2, \tau_4 \rangle$. It can complete τ_2 and τ_4. (*iii*): LLF first schedules task with least laxity. For τ_i, the laxity l_i is computed as $l_i = d_i - rc_i - t$. It can complete tasks τ_2 and τ_4 with scheduling sequence $\langle \tau_2, \tau_4 \rangle$.

For the given task set $\mathcal{T} = \{\tau_1, \tau_2, \tau_3, \tau_4\}$ at $t = 0$, all the three scheduling algorithms can complete two tasks. In this regard, the performance of these three algorithms is the same. However, all these results are obtained based on current knowledge of task without consideration of the impact of new arriving tasks. In practical environment, when there are new tasks arriving, the performance of the three algorithms may be different. Here, we come to a criterion.

Criterion. *A task set can be scheduled by different scheduling algorithms, when these algorithms can complete the same number of tasks, the one that can complete this number of tasks within less time slots makes better performance.*

Based on this criterion, SRTF is considered to make better performance than EDF and LLF. All of the three scheduling algorithms achieve two as the number of task completion. We wonder if it is the maximum number. For this simple example, we can enumerate all the subset of $\mathcal{T}$, and use EDF to tell if the subset of tasks is feasible. By this way, we can find three is the maximum number of task completion with scheduling sequence $\langle \tau_3, \tau_1, \tau_4 \rangle$. Through this example, we can see that, for overloaded real-time system, a new scheduling algorithm is needed. A novel greedy scheduling algorithm is proposed in next section.

III. Greedy Scheduling

For a given task set $\mathcal{T}$, assume the maximum number of tasks that can be successfully completed is m, and the corresponding can be completed task set is $\mathcal{T}^s \subseteq \mathcal{T}, |\mathcal{T}^s| = m$. If we can find $\mathcal{T}^s$, which is a feasible task set, using an optimum scheduling algorithm (e.g., EDF) to schedule $\mathcal{T}^s$ can make all the tasks in $\mathcal{T}^s$ meet deadlines, which means achieving the maximum number of task completion. Thus, the key problem is how to find the task set $\mathcal{T}^s$.

To find $\mathcal{T}^s$, a feasible procedure can be simply interpreted as: select each task from $\mathcal{T}$ based on a specified order, and use a method to judge if the selected task should be added into $\mathcal{T}^s$. Only a proper task indicated by the judgment method can be added into $\mathcal{T}^s$. There are two things that need to be decided: (*i*) the selecting order, (*ii*) the judgment method. In this paper, our proposed scheduling algorithm utilizes the idea of greedy algorithm to decide these two things. Greedy algorithm is a heuristic method that makes locally optimum choice at each step with the hope of finding a global optimum. The optimum choice is according to the optimum target. Based on our objective and the proposed criterion, we propose two optimum targets as follows: (*i*) maximizing the number of tasks in $\mathcal{T}^s$. (*ii*) when there are different choices that can put the same maximum number of tasks into $\mathcal{T}^s$, the one that can achieve the least value of $\sum rc_i$, for all $\tau_i \in \mathcal{T}^s$, should be chosen. Two theorems are proposed for every step of constructing $\mathcal{T}^s$.

Theorem 1. *In the procedure of constructing task set $\mathcal{T}^s$, if a task in $\mathcal{T}^s$ is replaced by another one with longer remaining execution time, it will conflict with the optimum targets.*

Proof. Assume a task τ_j in $\mathcal{T}^s$ is replaced by a task $\tau_i, rc_i > rc_j$, there are two situations: (*i*) if τ_i replaces τ_j only, the number in $\mathcal{T}^s$ is the same as before, but the value of $\sum rc_i$, for all $\tau_i \in \mathcal{T}^s$ will be more than before, this conflicts with the optimum targets. (*ii*) if τ_i replaces more than one tasks, the number of task completion will be less than before, this conflicts with the optimum targets too. □

Theorem 2. *For a feasible task set, if a scheduling algorithm allocates idle time slots to tasks backwards from their deadlines, regardless of the allocation order of the tasks, it can make all the tasks meet their deadlines, i.e., it is an optimal scheduling algorithm.*

Proof. It is trivial that a feasible task set $\mathcal{T}$ should meet the condition $\forall \mathcal{T}' \subseteq \mathcal{T}$, for all tasks $\tau_i \in \mathcal{T}'$, $\sum rc_i \leq \texttt{d\_max}(\mathcal{T}') - t$, where `d_max` returns the maximum value of d_i of τ_i in $\mathcal{T}'$. For a given task set $\mathcal{T}$, let's randomly choose two tasks τ_1, τ_2 from $\mathcal{T}$, and allocate rc_1 idle time slots to τ_1 backwards from d_1 first. As $\mathcal{T}$ is feasible, $rc_1 + rc_2 \leq \texttt{max}(d_1 - t, d_2 - t)$.

Algorithm 1 Greedy Scheduling (GS)

1: sort $\mathcal{T}$ by ascending order of rc_i, such that $\langle \tau_1, \tau_2, \ldots, \tau_n \rangle$ is a permutation of tasks in $\mathcal{T}$ with $rc_i \leq rc_{i+1}$ for all i, $1 \leq i < n$
2: $s := \langle\rangle$, $\mathcal{T}^s := \varnothing$
3: **for all** $1 \leq i \leq n$ **do**
4: **if** s.`CalIdle`$(t, d_i) \geq rc_i$ **then**
5: s.`BackAllocate`(τ_i, rc_i, d_i)
6: $\mathcal{T}^s := \mathcal{T}^s \cup \{\tau_i\}$
7: **end if**
8: **end for**
9: sort $\mathcal{T}^s$ by ascending order of d_i to construct scheduling list sl

1) if $d_2 \geq d_1$: we can get, $rc_1 + rc_2 \leq d_2 - t$. The available time slots for τ_2 is $d_2 - t - rc_1 \geq rc_2$. Thus, τ_2 can be allocated enough slots.

2) if $d_2 < d_1$: we can get, $rc_1 + rc_2 \leq d_1 - t$. (*i*) if $rc_1 \leq d_1 - d_2$, as τ_1 is allocated time slots backwards from d_1, thus the available time slots for τ_2 is $d_2 - t \geq rc_2$. (*ii*) if $rc_1 > d_1 - d_2$, the available time slots for τ_2 is $d_2 - t - rc_1'$, where rc_1' represents the time slots allocated to τ_1 in interval $[t, d_2)$. As τ_1 is allocated time slots backwards from d_1, $rc_1' = rc_1 - (d_1 - d_2)$, the available time slots for τ_2 is $d_1 - t - rc_1 \geq rc_2$. Thus, τ_2 can be allocated enough slots.

Then allocate idle time slots to the next task, with similar analysis, the next task also can be allocated enough time slots. Repeat allocating, all tasks in $\mathcal{T}$ can meet their deadlines. □

A. Scheduling Procedure

Theorem 1 gives the idea that we should select tasks with ascending order of rc_i. As to the judgment method, because we want to add tasks into $\mathcal{T}^s$ as many as possible, meanwhile ensure the task set $\mathcal{T}^s$ is feasible, based on theorem 2, the judgment method can be described as: when we add a task into $\mathcal{T}^s$, first try to allocate rc_i idle time slots backwards from d_i; only a task that can be allocated enough time slots should be added into $\mathcal{T}^s$. Based on these, we can find $\mathcal{T}^s$. Then, use EDF to schedule $\mathcal{T}^s$, we come to the procedure of GS constructing scheduling list. The detail is summarized in Alg. 1.

In list s (line 2) and sl (line 9), the i-th element is the index of a task that has been allocated i-th time slot. Function `CalIdle`(t, d_i) calculates the number of idle time slots in interval $[t, d_i)$. It returns $d_i - t - \Theta(t, d_i)$, where $\Theta(t, d_i)$ is the number of time slots which have been allocated to tasks in interval $[t, d_i)$. `BackAllocate`(τ_i, rc_i, d_i) allocates rc_i idle time slots to task τ_i backwards from d_i. Notice that, scheduling method that allocates idle time slots to tasks backwards from their deadlines is called *deferrable scheduling* (DS) [5]. Theorem 2 has proven that scheduling algorithm which uses the deferrable scheduling method is an optimum scheduling algorithm. In GS, the worst-case time complexity of `CalIdle` and `BackAllocate` are both $O(d_{max})$, where d_{max} is the maximum d_i for $\forall \tau_i \in \mathcal{T}$. If we use quick sort algorithm to sort $\mathcal{T}$ and $\mathcal{T}^s$, the worst-case time complexity of sort operation is $O(n^2)$. Thus, GS has a complexity that is pseudo-polynomial: $O(n \cdot d_{max} + n^2)$.

Recall the example depicted in Fig. 1. GS can successfully complete three tasks with the optimum scheduling sequence $\langle \tau_3, \tau_1, \tau_4 \rangle$. Each time GS scheduling tasks makes an optimum choice for currently known task set. However, as scheduler has no knowledge of a task until it arrives in the system, it is doubtful that GS can make an optimum choice for the set of overall tasks.

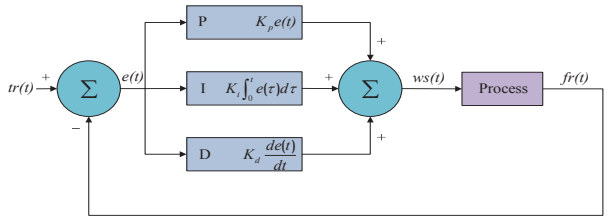

Fig. 2. Window Size (ws) controlled by PID controller

IV. Greedy Scheduling with Feedback Control

GS selects proper tasks from $\mathcal{T}$ and adds it into $\mathcal{T}^s$. It expects tasks in $\mathcal{T}^s$ can all be completed. Nevertheless, when system is overloaded, the selected proper tasks usually cannot all be successfully completed. This observation gives the idea that the capacity of $\mathcal{T}^s$ (i.e., the maximum number of tasks that can be added into $\mathcal{T}^s$) should be limited based on task completion condition in $\mathcal{T}^s$. Here, we use window size, represented by ws, to denote this capacity.

As $\mathcal{T}^s$ keeps on changing, its snapshot $\mathcal{T}^{s'}$ is used to be the observed task set. When we take a snapshot of $\mathcal{T}^s$ at system time t, its current value is assigned to $\mathcal{T}^{s'}$. A new snapshot is taken when the completion conditions (completed or discarded) of all the tasks in previous snapshot are determined. We use failure ratio, represented by fr, to denote the ratio of unsuccessfully completed tasks in $\mathcal{T}^{s'}$. The goal of controlling ws is not just to make all of the tasks in $\mathcal{T}^s$ meet their deadlines (i.e., with fr of 0.00), but to achieve this goal with the largest possible task population in $\mathcal{T}^s$. For this reason, fr's target value tr is set to 0.05.

When system schedules tasks, fr should be kept on monitoring. Based on its monitored value, ws can be dynamically changed to justify fr to reach its target value tr. To achieve this, one feasible method is introducing a feedback controller. By this way, GS is extended to GSFC. To the consideration of simplicity, a proportional-integral-derivative (PID) controller is used. Although it is very simple, its effectiveness has been demonstrated in industrial control systems. It is also believable that if PID controller can achieve good performance, a more sophisticated controller can achieve better. The block diagram of PID controller is shown in Fig. 2. The variable e represents the tracking error which is the difference between variable fr and the desired target value tr. At system time t, the PID controller attempts to minimize the absolute value of e by three computing terms: proportional, integral, and derivative, weighted by tunable gains K_p, K_i, and K_d, respectively. The computed result is to set ws which can affect the value of fr. As ws represents the capacity of $\mathcal{T}^s$, due to its practical meaning, ws is defined as an integer variable with a lower bound 1.

V. Performance Evaluation

In this section, we present the results of simulations which are conducted to study the performance of different scheduling algorithms. The scheduling algorithms that are used to compare with GS and GSFC are SRTF, EDF, LLF, and their corresponding DS methods, i.e., DS-SRTF, DS-EDF, DS-LLF. The DS method is introduced in section III-A. The difference of scheduling procedure among DS-SRTF, DS-EDF, and DS-LLF is the order of selecting tasks. DS-SRTF selects task with

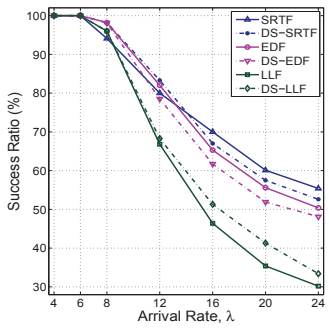

Fig. 3. Comparison of baseline algorithms ($4 \leq \lambda \leq 24$)

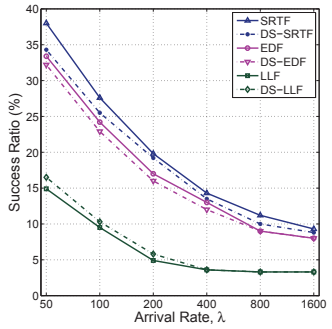

Fig. 4. Comparison of baseline algorithms ($50 \leq \lambda \leq 1600$)

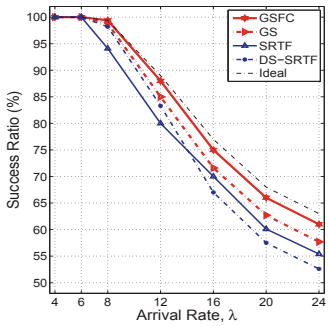

Fig. 5. Performance of GSFC and GS ($4 \leq \lambda \leq 24$)

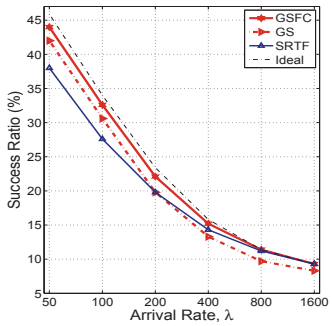

Fig. 6. Performance of GSFC and GS ($50 \leq \lambda \leq 1600$)

ascending order of rc_i, while DS-EDF and DS-LLF select task with ascending order of d_i and task laxity, respectively.

A. Simulation Settings

The metric used to evaluate the scheduling performance is *success ratio* which is the percentage of tasks that have been successfully completed. The setting of total number of input tasks is 1000. The input tasks are generated according to uniform distribution with arriving rate λ which represents the number of tasks that arrive in the system per 100 time units. For each task τ_i, c_i varies uniformly in [1 25]. The assignment of d_i is according to the equation: $d_i = r_i + sf_i * c_i$, where sf_i is the slack factor that indicates the tightness of task deadline. For each task τ_i, sf_i varies uniformly in [1 16]. The tuned values for the gains of PID controller K_p, K_i, K_d are 5, 0.017, 12, respectively. As the existing of lower bound for ws, the controller takes the measure of anti-windup.

B. Results and Analysis

As the change rate of success ratio is quit different in different intervals of λ, the results are shown separately in two intervals: [4 24] and [50 1600]. Beside comparing with the baseline algorithms, we also want to know how far the performance of GSFC is from the performance upper bound in terms of success ratio. If the controller in GSFC could set ws perfectly to make sure that every time GSFC *just* completes all the tasks in $\mathcal{T}^s$ and no processor time slot is wasted in executing unsuccessfully completed tasks, the GSFC could achieve the ideal performance. In order to get the performance upper bound, we manually manipulate the value of ws. For the specific input task sets, we can get the ideal results which are represented by the lines "Ideal" in Fig. 5 and Fig. 6.

TABLE I. TOTAL TIME CONSUMPTION FOR SCHEDULING ALL TASKS

λ	GSFC (ms)	GS (ms)	SRTF (ms)	DS-SRTF (ms)
4	286	113	24	70
24	1002	828	82	471
50	1944	1339	156	806
1600	2824	2569	351	955

As shown in Fig. 3 $\sim$ Fig. 6, compared with the baseline algorithms, GS performs best when $\lambda \leq 200$. As when $\lambda > 200$, the success ratio is around 20%, which means system is severely overloaded. This condition rarely happens in practical environment. Thus, we can say GS achieves better overall performance than all the baseline algorithms. For GSFC, it achieves the best performance under all the different workload conditions. Compared with GS, this observation proves the effectiveness of feedback control. The improvement appears when $\lambda \geq 8$. This is because when $\lambda < 8$, the overload condition is not serious. The capacity of $\mathcal{T}^s$, computed by PID controller, is larger than the number of tasks that can be added into $\mathcal{T}^s$. This makes these two methods have the same performance. When $\lambda \geq 800$, it can be seen that GSFC and SRTF achieve the same best performance. The reason is that, under such serious overload condition, the capacity of $\mathcal{T}^s$ equals to its lower bound 1 at most of the time, it makes the GSFC act similarly as SRTF.

The total time consumption for scheduling all the input tasks are shown in Table I. As the computing procedure of GSFC is more complicated than other algorithms, the time consumption of GSFC is larger than others. However, we should notice that, the time consumption is for scheduling all the input tasks (1000 input tasks in the simulation). It is quite a good deal for system to spend a little longer time (usually less than 2 seconds) on scheduling to make much more tasks (around 100 compared with DS-SRTF) meet their deadlines.

VI. Concluding Remarks

The design of scheduling algorithm is crucial for overloaded real-time systems. In this paper, we focus on maximizing the total number of tasks that meet their deadlines. To achieve this objective, a novel scheduling algorithm GSFC was proposed. As shown in the performance studies, it can effectively improve system performance. For the future work, an important direction is to use more sophisticated feedback controller to further improve the performance of GSFC.

References

[1] P. Derler, E.A. Lee, and A.S. Vincentelli, "Modeling Cyber-Physical Systems," *Proc. IEEE*, vol. 100, no. 1, pp. 13–28, Jan. 2012.

[2] S.K. Baruah, J. Haritsa, and N. Sharma, "On-line Scheduling to Maximize Task Completions," *Proc. 15th IEEE Real-Time Syst. Symp.*, pp. 228–236, Dec. 1994.

[3] J.R. Haritsa, M.J. Carey and M. Livny, "On Being Optimistic about Real-Time Constraints," *Proc. 9th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Syst.*, pp. 331–343, Apr. 1990.

[4] F. Zhang and A. Burns, "Schedulability Analysis for Real-Time Systems with EDF Scheduling," *IEEE Trans. Comput.*, vol. 58, no. 9, pp. 1250–1258, Apr. 2009.

[5] M. Xiong, S. Han, K.-Y. Lam, and D. Chen, "Deferrable Scheduling for Maintaining Real-Time Data Freshness: Algorithms, Analysis, and Results," *IEEE Trans. Comput.*, vol. 57, no. 7, pp. 952–964, July 2008.