Work-in-Progress: A Unified Runtime Framework for Weakly-hard Real-time Systems

Hyunjong Choi and Hyoseung Kim
University of California, Riverside
hchoi036@ucr.edu, hyoseung@ucr.edu

***Abstract*—A weakly-hard real-time system is a system that can tolerate a bounded number of timing violations. There have been various assumptions made by prior work on handling deadline-missed instances (jobs) of a task in weakly-hard systems, e.g., terminate it immediately or continue to execute it even after its deadline is missed. However, no prior work has presented a system-level framework to support such options and to quantitatively assess their effects on schedulability. In this paper, we present a *unified runtime framework* to execute weakly-hard tasks with the four major deadline-miss handling options: job abort, delayed completion, job pre-skip, and job post-skip. Our work is applicable to any type of operating systems that support preemptive scheduling with task control blocks. We have implemented our framework in the Linux platform running on Raspberry Pi. We evaluate the performance of each scheme and measure spatial and computational overheads.**

I. Introduction

Weakly-hard real-time systems have been studied to capture the occasional miss of deadlines that a system can tolerate. The common notation of a weakly-hard constraint is in the (m, K) form, which specifies that at most m instances can miss their deadlines among K consecutive instances. A certain level of quality of service can be guaranteed, provided that such a timing violation happens in a known and predictable way.

In the literature of weakly-hard systems, there are various assumptions on how to handle a task's instance (job) when it has missed or is likely to miss the deadline. A job may be terminated immediately when it misses its deadline, i.e., *job abort*. It may continue to execute to completion even if its deadline has passed, i.e., *delayed completion*. A prediction can be made such that only the jobs that are expected to complete by their deadlines are executed, i.e., *job pre-skip*. One may also decide to skip the next job if the current job is executing over the next period, i.e., *job post-skip*. Table I summarizes the classifications of previous weakly-hard studies based on the above assumptions.

Prior work, however, has not considered a runtime framework to instantiate various deadline-miss handling options and has not provided any comparative analysis among those assumptions. Besides, an implementation cost, which is one of the criteria to evaluate applicability in practical systems, has not been thoroughly studied.

This paper presents our work-in-progress effort on developing a runtime framework for weakly-hard real-time systems. Our framework includes systems primitives to support the four aforementioned deadline-miss handling schemes, i.e., *job abort*, *delayed completion*, *job pre-skip*, and *job post-skip*. The current version of our framework focuses on task-level fixed-priority scheduling and has been prototyped in the Linux kernel on Raspberry Pi 3. The proposed framework design approaches can also be applied to any operating system (OS) that uses preemptive task scheduling with task control blocks. We expect that our framework can serve as a basis to build real-time applications with weakly-hard constraints and facilitates the comparison of different weakly-hard schemes on a real platform.

TABLE I: Weakly-hard studies based on the job handlings

Handling scheme	Prior work
Job abort	[6]*, [7]*, [3]
Delayed completion	[5], [8]
Job pre-skip	[6]*, [7]*

**: multiple handling schemes are employed.*

II. Related work

Many weakly-hard studies have assumed the use of the delayed completion scheme, in which a job continues to run although it exceeds its deadline. In overloaded systems, the work in [5] bounds temporary violations of deadlines by using typical worst-case analysis (TWCA) and job arrival curves. The work in [8] also uses the delayed completion scheme and focuses on a taskset with utilization less than or equal to 1.

In [3], the author assumes that the execution of a job is aborted if it does not finish within its deadline. This assumption is to ensure that a delayed job from the previous period does not affect the execution of the next released job. With this assumption, tasksets with utilization more than 1 can be schedulable, but the author has not provided details on the safe recovery of task states after the job abortion.

To manage the overloaded situations, some prior work [6, 7] has considered both the job abort and the job pre-skip schemes. Ramanathan [7] classified jobs into mandatory and optional ones such that only the mandatory jobs are guaranteed to be schedulable in order to reduce the overall loads of the system. The work in [6] also used similar classification approaches, whereas the optional jobs may be executed by checking its eligibility based on slack or predetermined patterns. Recently, Zhishan et al. [4] proposed a new scheme for mixed-criticality systems that drops jobs while guaranteeing the predefined level of performance degradation of low-criticality tasks.

III. System Model

Our framework primarily considers task-level fixed-priority preemptive scheduling in a uniprocessor system. For the task model, we assume periodic tasks with weakly-hard constraints.

Task model. Task τ_i is represented as follows:

$$\tau_i := (C_i, D_i, T_i, (m_i, K_i))$$

- C_i: The execution time of each job of a task τ_i.
- D_i: The relative deadline of each job of τ_i ($D_i \leq T_i$).
- T_i: The period of τ_i.
- (m_i, K_i): The weakly-hard constraints of τ_i ($m_i < K_i$). If τ_i is a hard real-time task, $m_i = 0$ and $K_i = 1$.

The j-th job of a task τ_i is denoted as $J_{i,j}$.

Performance metrics. To evaluate the performance of weakly-hard schedulers with different deadline-miss handling schemes, we define the following metrics.

Def. 1. *The effective utilization of a task τ_i, $U_i^e(t)$, measures the ratio of the time used for jobs that have met their deadlines during a given time interval t. Hence, $U_i^e(t) = \frac{C_i \times M_i}{t}$, where M_i is the number of jobs completed by deadline during t.*

The total effective utilization, $U^e(t)$, is thus the sum of the effective utilization of all tasks in the system during a given time interval t, i.e., $U^e(t) = \sum_{i=1}^{N} U_i^e(t)$, where N is the number of tasks and $U^e(t)$ cannot exceed 1.

Def. 2. *The runtime utilization of a task τ_i, $U_i^r(t)$, measures the ratio of the time used for a task that has occupied the processor during a given time t. Hence, $U_i^r(t) = \frac{R_i(t)}{t}$, where $R_i(t)$ is the processor time used by a task τ_i during t.*

The total runtime utilization, U^r, is the sum of the runtime utilization of all tasks in the system during time t, i.e., $U^r(t) = \sum_{i=1}^{N} U_i^r(t)$, and it cannot exceed 1. Also, $U^r(t) \geq U^e(t)$.

IV. Runtime Framework

This section presents our framework to support the four deadline-miss handling schemes mentioned in Section I: *job abort*, *delayed completion*, *job pre-skip* and *job post-skip*. We begin with a fundamental runtime mechanism for periodic execution of tasks, and then present the detailed design of each scheme based on it.

A. Periodic execution support

Fig. 1 illustrates the overview of our runtime mechanism for periodic task execution, which is organized as *user space* and *kernel space*. In the user space, a task can request specific actions to the OS kernel as needed, e.g., registering a task as a real-time task or putting it in sleep mode when the current job finishes execution. In the kernel space, our runtime consists of four core modules: initializer, scheduler, timer, and complete sequence. We detail the functions of each module as follows:

- *Initializer*: The system registers a task as a periodic real-time task by assigning real-time priority, and creating release and deadline timers for this task in Ⓐ.

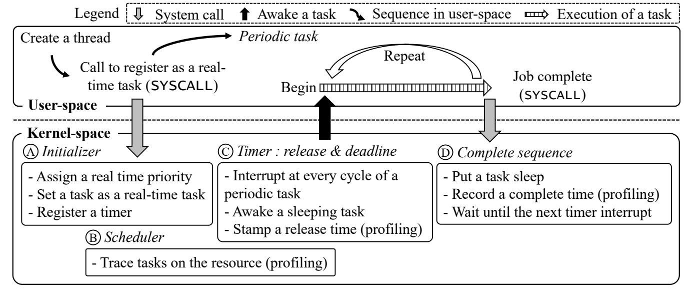

Fig. 1: Runtime mechanism for periodic task execution

- *Scheduler*: In Ⓑ, a scheduler determines the *readiness* of periodic tasks based on the deadline-miss handling scheme in use, and records task execution traces and resource usage.
- *Timer*: There are two timers involved for each periodic task. First, the release timer (Ⓒ) fires at the beginning of each period of the task. The deadline timer fires when the task has not completed its job execution and is still in running mode, different sequences are followed based on the deadline-miss handling used.
- *Job completion sequence.*: Once the task finishes its job execution, it makes a `job_complete` system call to the kernel (Ⓓ). The system then puts the task into sleep mode so that it waits until the next job is released by the timer.

Note that this mechanism shown in Fig. 1 is generally applicable to most OSs supporting preemptive task scheduling.

B. Job abort

In this scheme, a job is terminated immediately if it misses its deadline. This scheme can be beneficial in the case where a deadline-missed job will no longer need to run as the remainder of its execution does not yield any gain.

However, the termination of a running job is not trivial in the implementation. When a task misses its deadline, it needs to be rolled back to its previous state so that its next job can safely and correctly execute in the next period. This rollback mechanism is typically done by creating a checkpoint [1].

Task rollback. There are two types of rollback approaches we may consider. The first is task-level checkpointing, where each task creates its own checkpoints as part of execution and implements a handler to recover from the stored checkpoints. The second approach is system-level checkpointing, where the OS or middleware creates each task's checkpoint, e.g., by storing all memory pages recently modified, and recovers the task state when needed. In this work, we focus on the task-level approach due to its lower overhead that can be done in three steps as follows:

- **Step 1. Store a checkpoint**: Since our rollback technique is achieved in a task-level, a checkpoint is stored at the beginning of a task execution. We used `sigsetjmp` in the standard C library to save a program counter (PC) and a stack pointer (SP)
- **Step 2. Notify a deadline miss to the user space**: If the job has missed its deadline, the kernel notifies the task by sending a signal.

- **Step 3. Recover from the checkpoint**: By the signal generated in Step 2, the signal handler of the task is triggered so that the PC and SP are recovered from the stored checkpoint by using `siglongjmp`.

TABLE II: Taskset 1

Tasks	T (Period) [ms]	C (WCET) [ms]	(m, K)	Priority
τ_1	65	35	(2,4)	High
τ_2	125	35	(2,4)	Middle
τ_3	200	35	(2,4)	Low

Example. We have implemented this scheme in the Linux kernel v4.9.76 running on Raspberry Pi 3, and tested the operation of the job abort scheme by running a taskset given in Table II. In Fig. 2, τ_1 and τ_2 are always schedulable while $J_{3,1}$ and $J_{3,4}$ of Task 3 do not meet their deadlines.

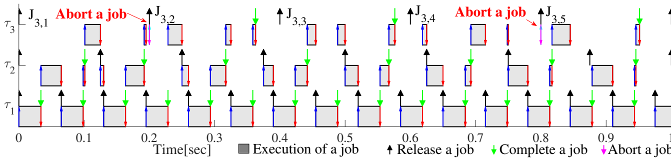

Fig. 2: Job abort

C. Delayed completion

The delayed completion scheme allows a deadline-missed job to continue to run until it completes. This scheme is effective when the quality of service can be improved by the execution of remainder of a deadline-missed job.

Under this scheme, however, if a job continues to run over its deadline, the next released job is delayed by the execution of the previous job. Thus, this scheme does not get benefit from the weakly-hard concept in overloaded situations, i.e., taskset with total utilization is more than 1.

In order to realize the scheme, the task is put in sleep mode only when the job just completed is the latest released job. This is checked in by the job completion sequence module (Ⓓ in Fig. 1).

Example. As shown in Fig. 3, deadline-missed jobs are running until it completes its execution under this scheme.

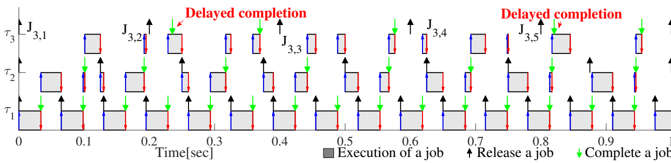

Fig. 3: Delayed completion

D. Job pre-skip

The job pre-skip scheme determines whether to execute a job or not at its release time (the release timer Ⓒ of Fig. 1). A decision for the execution can be made in either online or offline. In case of the online approach, the system can use the slack time of a task at the moment of its job release. In the offline approach, a predetermined execution pattern is used, e.g., `1010` where `1` means execution and `0` means skip.

However, there are two major drawbacks. The first is the runtime overhead which may be high because the scheduler needs to check the slack time (online) for job execution. Moreover, especially when the average-case execution time is much lower than the WCET, the scheduler may unnecessarily skip jobs, which results in processor underutilization.

In this scheme, the release timer (Ⓒ) is the major module to be modified to enable the pre-skip scheme. The overall sequence of the modified release timer is depicted in Fig. 4.

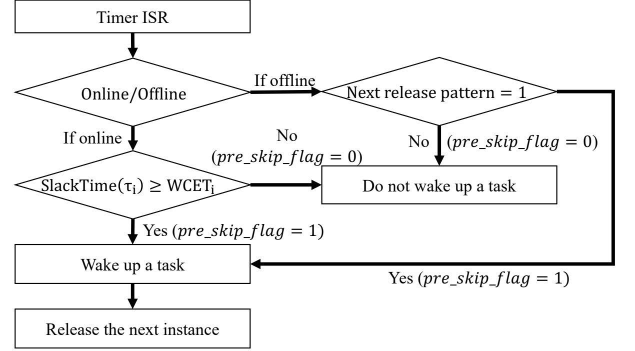

Fig. 4: Timer sequence in job pre-skip scheme

Example. Under the offline approach, a predefined pattern of `1010` is applied to all tasks so that every other instance is executed as depicted in Fig. 5. On the other hand, under the online approach, jobs $J_{3,2}$, $J_{3,3}$, and $J_{3,5}$ are executed based on slack calculation.

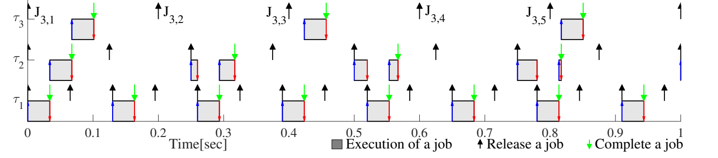

Fig. 5: Job pre-skip (pattern)

E. Job post-skip

In this scheme, the scheduler allows a deadline-missed job (released at j^{th}) continue to run, but it always skips the next released job (released at $j+1^{th}$) and keeps track of the index of the job. Under this scheme, jobs are discarded occasionally, resulting in degradation of the quality of service of a system.

Example. Under the job post-skip scheme, $J_{3,2}$ and $J_{3,5}$ are skipped because the previous jobs have violated their deadlines and affected the execution of the next released jobs, as shown in Fig. 6.

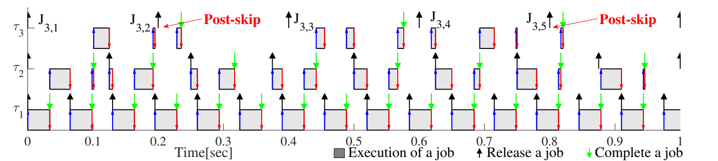

Fig. 6: Job post-skip

V. Evaluation

In this section, we evaluate the proposed framework in the Linux kernel running on Raspberry Pi 3 (Quad Cortex

A53 @ 1.2GHz). The evaluation consists of two parts: the measurement of computational overheads and the case study.
Overheads. For the computational overhead of our framework, it is worth noting that the delayed completion and post-skip schemes do not cause any additional cost other than those for the periodic execution mechanism shown in Section IV-A. However, under the job abort and job pre-skip schemes, there are the following four major sequences that can cause extra runtime overhead:

- `sigsetjmp` (job abort): the cost in the user space to create a checkpoint
- `siglongjmp` (job abort): the cost in the kernel space to send a signal to the user-space task
- Slack (job pre-skip): the cost of calculation of the slack
- Pattern (job pre-skip): a transition cost of pointing to the next element in the pattern

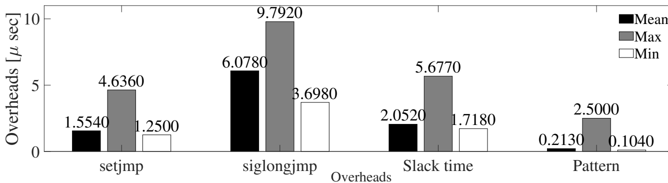

Fig. 7: Overheads of taskset

Fig. 7 summarizes the overhead measurement on Raspberry Pi 3. As can be seen, `siglongjmp` for the rollback mechanism is the most costly operation. However, they are acceptably small in μs units, compared to the WCET and periods typically denoted in ms units.
Case study. For case study, we have selected a taskset given in Table III where its total utilization is higher than 1.

TABLE III: Taskset 2 [6]

Tasks	T (Period) [ms]	C (WCET) [ms]	Skip parameter
τ_1	6	1	2
τ_2	7	4	2
τ_3	19	5	2

This taskset is not schedulable under any conventional fixed-priority scheduler. However, if the RM-RTO[1] algorithm of [6] is used, the job execution pattern of each task is determined as either `10` or `01` and this pattern can be realized by the job-skip scheme. Fig. 8 shows the result of dynamic failures[2] of τ_3.

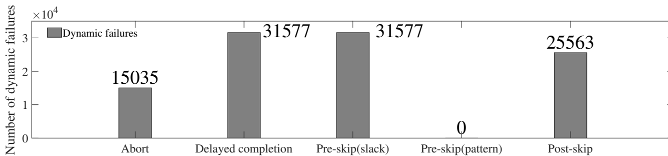

Fig. 8: Dynamic failures of τ_3 (total 31578 jobs released)

As can be seen, the number of dynamic failures under the pre-skip scheme (pattern) is zero, meaning that τ_3 satisfies its weakly-hard constraint. However, all the other schemes suffer from a high number of dynamic failures.

[1]RM-RTO stands for Rate Monotonic Red Task Only.

[2]A task experiences more than m deadline misses in a window of K jobs.

Fig. 9 shows the observed effective and runtime utilization values. The abort scheme shows the highest effective utilization and the pre-skip scheme (pattern) has the lowest. This is because the pattern-based scheduling pessimistically skips the execution of higher-priority tasks even if there is enough processor time to use.

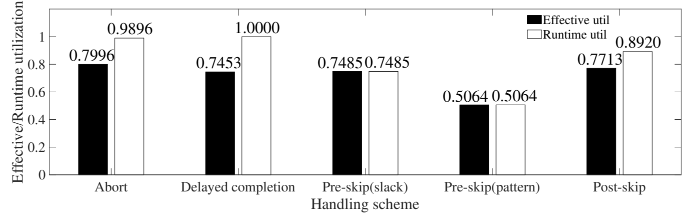

Fig. 9: Total effective and runtime utilization

VI. Conclusion

We proposed a unified runtime framework for multiple deadline-miss handling schemes in weakly-hard real-time systems. The framework has been implemented in the Linux kernel on Raspberry Pi with very low overhead, but it is easily applicable to other OSs using fixed-priority preemptive schedulers. Experimental results show that, depending on the deadline-miss handling scheme used, the number of violations of weakly-hard constraints as well as utilization metrics can vary significantly for the same taskset and a different trend can be observed for other tasksets. These results pave an interesting research direction to investigating weakly-hard tasks under diverse experimental conditions and new analysis techniques. In our current implementation, all tasks in the system are governed by the same deadline-miss handling scheme. This is in accordance with the assumptions of prior work, but we expect that allowing each task to have a different handling scheme would increase resource efficiency and design flexibility. Furthermore, our framework can be extended beyond task-level fixed-priority scheduling, e.g., a job-class-level fixed-priority scheduler [2] to be presented in RTAS 2019. It can also be used as an assessment tool for the issues that have not studied much in the weakly-hard context such as an inter-task dependency, shared resources, multicore systems, and temporal interference from contention in cache and main memory.

References

[1] M. Asberg et al. Resource sharing using the rollback mechanism in hierarchically scheduled real-time open systems. In *IEEE RTAS*, 2013.

[2] H. Choi, H. Kim, and Q. Zhu. Job-Class-Level fixed priority scheduling of weakly-hard real-time systems. In *IEEE RTAS*, 2019.

[3] J. Goossens. (m, k)-firm constraints and DBP scheduling: impact of the initial k-sequence and exact schedulability test. 2008.

[4] Z. Guo et al. Uniprocessor mixed-criticality scheduling with graceful degradation by completion rate. In *IEEE RTSS*, 2018.

[5] Z. A. H. Hammadeh et al. Budgeting under-specified tasks for weakly-hard real-time systems. In *ECRTS*, 2017.

[6] G. Koren and D. Shasha. Skip-over: Algorithms and complexity for overloaded systems that allow skips. In *RTSS*, 1995.

[7] P. Ramanathan. Overload management in real-time control applications using (m, k)-firm guarantee. *IEEE Trans. on Par and Dist. Syst.*, 10(6):549–559, Jun 1999.

[8] Y. Sun and M. D. Natale. Weakly hard schedulability analysis for fixed priority scheduling of periodic real-time tasks. *ACM TECS*, 2017.