Stealing Static Slack via WCRT and Sporadic P-Servers in Deadline-Driven Scheduling

Zhishan Guo*¶, Sudharsan Vaidhun†¶, Abdullah Al Arafat*¶, Nan Guan‡, Kecheng Yang§
*North Carolina State University, †University of Central Florida, ‡City University of Hong Kong, §Texas State University
{zguo32, aalaraf}@ncsu.edu, sudharsan.vaidhun@knights.ucf.edu, nanguan@cityu.edu.hk, yangk@txstate.edu

***Abstract*—Real-time systems are characterized by strict timing constraints represented by deadlines. Some systems are tight, such that jobs finish their execution right at the deadlines in the worst case, while others may not be so tight. Static slack is a concept that captures such non-tightness, and it can often be "stolen" to handle additional aperiodic job requests, task suspensions, and occasional task overruns. This paper identifies an interesting and direct correlation between worst-case response time (WCRT) and static slack in a deadline-driven uniprocessor system. We propose a systematic approach for safely constructing a set of Sporadic P-Servers to tightly capture the available static slack, given any feasible task set under a preemptive earliest deadline first. These P-Servers are special in that each task has only a unit-length execution budget and runs in a discrete manner. To leverage these P-Servers and "steal" the slack, we propose a novel consume-replenish algorithm to handle online hard aperiodic jobs. We also extend the theory for other applications, such as dealing with early and arbitrary self-suspensions and servicing job overruns in mixed-criticality systems without triggering a mode switch. Experiments demonstrate that the proposed theory can provide new and better schedulability in some subcases for each application.**

***Index Terms*—Slack Stealing, WCRT, Sporadic Server, EDF**

I. Introduction

Recurring workloads with real-time requirements have been traditionally modeled using task models such as the Liu and Layland model [19]. These task models are designed to capture the worst-case execution requirements of the workload. Similarly, the assumptions on the platform are based on worst-case availability. For example, when the clock speed of a microcontroller is susceptible to changes in the device's operating temperature, the slowest clock speed is considered and modeled. For a given workload-platform combination, there can be unreserved resources referred to as *static slack*. For example, if a workload requires 80% utilization of a fixed-speed uniprocessor, then the remaining 20% of the utilization capacity can be quantified as static slack—this paper focuses on how to calculate, model, and leverage them. On the other hand, *dynamic slack* refers to the unused portion of the reserved resources during execution—which is out of the scope of this work.

As models and workloads grow in complexity, static slacks play an increasingly vital role in modern systems' efficiency. Such an important role might not be obvious as workload and system models are proposed with "fancier" names. Below we highlight some well-known models in real-time scheduling theory by discussing their relationships with static slack.

a) Dealing with Online Aperiodic Jobs. Modern systems increasingly interact with the world and may need to react to unexpected situations via pre-designed components. As a result, it can be very helpful to design a system that can admit occasional additional workload requirements during run time. Static slack is naturally correlated with one-time *Aperiodic* job requests. With careful models describing static slack, correctness guarantees can be made to those aperiodic jobs if their behaviors (e.g., worst-case execution time) are known.

b) Self-Suspending Tasks. In the era of the Internet of Things and Deep Learning, real-time tasks communicate with external devices (such as solid-state/ magnetic disks and network cards) or leverage accelerators (such as GPUs). Such interactions often introduce self-suspension delays. In general, modeling such 'unavailability' with additional execution requirements can lead to significant resource capacity waste [9]. A moment's thought would convince the reader: multiple tasks may simultaneously suspend but can never execute together upon a uniprocessor platform. Unfortunately, classical worst-case response time and schedulability analyses, such as the critical instant theorem [19], Time-Demand Analysis [17], or the demand bound function [4], do not apply to real-time workloads with self-suspension, and has been demonstrated to be prone to flaw [9]. The study of static slack can be very powerful in handling some sub-cases for self-suspension tasks. We can tolerate early self-suspensions from all tasks with per-task static slack, while with system-level static slack, we may handle dynamic self-suspensions.

c) Handling Task Overrun without Dropping any Task in Mixed-Criticality Systems. As proposed by Steve Vestal, mixed-criticality systems [25] have two parallel components to its temporal correctness guarantees: the correctness criteria (of the workload) has to be met in low criticality mode as well as in high criticality mode. Generally, in the high criticality mode, there is some additional workload to be scheduled and with hard deadlines. Unlike most existing work in mixed-criticality that sacrifices correctness guarantees to less important workload (in the high criticality mode), we could maintain the execution budget for all workloads even when some critical tasks overrun by considering the problem in terms of static slack. For instance, a portion of the available static slack in the low criticality mode can be utilized to satisfy the additional workload without affecting the correctness of

¶These authors contributed equally

any task in the system.

Since static slack can be leveraged as a common resource for multiple important aspects of real-time systems, it is worth investigating. Despite several foundational works on calculating static slack [11], [24], [10], this work adds a novel and unique approach by studying its relationship with WCRT directly from a per-task perspective. From a system-wide perspective, we propose a novel Sporadic P-Server that calculates the minimal guaranteed static slack for a schedulable workload under Earliest Deadline First (EDF) and describe techniques to apply such a server to handle additional workloads or suspensions.

Section II formally describes the models and notations, after which we answer the following research questions in Section III: (i) Given any non-tight EDF-schedulable task set (where jobs always finish before the deadlines), can we systematically calculate a minimum available static slack? (ii) If so, what could be the slack availability pattern—Is it per-task or per-system (hyperperiod)? Is it periodic or aperiodic? (iii) What would be the usability conditions of slack; i.e., can one leverage slack at any time in the window?

Specifically, the contributions of Section III are two folds: (i) we establish a link between EDF WCRT analysis and per-task available static slack[1] in the beginning part of its scheduling window, and prove its correctness; (ii) we propose to capture static slack as a function of time in a novel sporadic P-Fair-style server form named Sporadic P-Server, and establish the corresponding server calculation algorithm and prove its correctness and tightness.

Section IV further demonstrate several use-cases for static slacks in (i) how to consume the budget of Sporadic P-Server and how to replenish in order to handle the aperiodic jobs with hard deadlines; (ii) handling task self-suspensions both offline (restricted to early suspension only that appears before calculation) and online (with known total suspension length per task but arbitrary suspension window numbers and locations); and (iii) dealing with task overrun without dropping workload in mixed-criticality systems.

Since the proposed slack stealing approach is a 'one-size-fits-all' solution for multiple problems, we do not expect it to outperform the state-of-the-art methods in each of the application problems separately in general. However, for several sub-cases with certain additional restrictions, the proposed approach outperforms state-of-the-art methods in each aforementioned application domain. Section V describes these experimental comparisons in sub-cases of those applications to show the applicability and flexibility of the proposed approach. Section VII concludes the work and points out several future research directions.

[1]Note that per-task static slack (in short, static slack, or slack) in this paper is different from per-job laxity/slack in the existing literature. The static slack is an offline worst-case parameter and must be safe for "stealing" from all jobs' perspectives, not only for the job/task being considered. For example, in the EDF schedule of the task set in Table I shown in Figure 2, Slack of τ_2 is 2. Although it has a laxity or online slack of 3 at time zero, there can be deadline miss (for τ_1) if we set τ_2 to be ineligible for three-time units upon release, although $\tau_{2,j}$ has 3 units of laxity for any j.

II. Model, Notations, and Preliminaries

For the real-time scheduling problems considered in this work, we use a $2-$tuple $\langle \mathcal{J}, \mathsf{Sched} \rangle$, where the first term denotes the workload and the second term denotes the scheduling algorithm used. Collectively, the $2-$tuple is a valid system if $\mathcal{J}$ is schedulable by the Sched. Note that we consider the workload to be a set of jobs while they can be 'released' by a set of tasks. This paper focuses on a uniprocessor system with preemptive schedulers.

A. Workload and Optimality of EDF

Consider a job set $\mathcal{J}$ with N jobs, each job $J_k \in \mathcal{J}$ is represented by a 3-tuple $J_k = (r_k, d_k, C_k)$, where r_k is the release time of the job, d_k is the absolute deadline of the job (by which it must finish its execution), and C_k is the WCET.

In real-time systems, pieces of code are often repeatedly executed. Thus we also consider sporadic task set $\mathcal{T}$ with n tasks, where each task $\tau_i \in \mathcal{T}$ is represented by a 3-tuple

$$\tau_i = (C_i, T_i, D_i) \tag{1}$$

where C_i is the WCET of the task, T_i is the minimum inter-arrival time between consecutive job releases, and D_i is the relative deadline of the task. The hyperperiod of the task set is H. The deadlines are assumed to be constrained deadlines, i.e., $D_i \leq T_i$, $\forall i$.

Since the focus of this paper is to 'steal' slack from an existing system, we are only interested in feasible task sets. According to Theorem 1 on the optimality of EDF, we assume the considered workload is schedulable by (preemptive) EDF scheduling algorithm upon a uniprocessor platform. In plain words, Theorem 1 tells us for any workload (with no suspensions or predecessor constraints), if there is a way to construct a correct schedule, then it would be schedulable under preemptive EDF.

Theorem 1. [19] [15] *When preemption is allowed and jobs do not contend for resources, the EDF algorithm can produce a feasible schedule of a set $\mathcal{J}$ of jobs with arbitrary release times and deadlines on a uniprocessor* if and only if *$\mathcal{J}$ has feasible schedules.*

Remark 1. *Note that $\mathcal{J}$ is any feasible set of jobs—these jobs could be generated by periodic or sporadic tasks (with implicit, constrained, or arbitrary deadlines), or they could be 'one-shot' jobs without a notion of recurrences. However, suspensions are not allowed.*

Remark 2. *Note that deadline tie-breaking rules do not affect the optimality of EDF. However, for consistency and easier applicability, this paper assumes that jobs with smaller task index values will be prioritized when jobs of the same absolute deadline compete for resources.*

During run-time, the tasks represented by the task model release a potentially infinite sequence of jobs. However, within a hyperperiod, the total number of jobs is finite. We use $\tau_{i,j}$ to denote the j^{th} job released by task τ_i, released at time $r_{i,j}$, and its absolute deadline becomes $d_{i,j} = r_{i,j} + D_i$.

TABLE I: Workload considered in Example II.1

	C_i	T_i	D_i	R_i	$S_i = D_i - R_i$
τ_1	1	3	3	1	2
τ_2	2	5	5	3	2
τ_3	1	10	8	5	3

At the task level, the WCRT R_i of any task τ_i indicates how long in the worst case ($\forall j$) it would take for $\tau_{i,j}$ to finish its execution from its release. By worst, we mean under all release patterns and among all the jobs released by this task. The WCRT can be calculated using EDF response time analysis proposed by Spuri [22]. Techniques such as QPA [26] can also be adapted to calculate WCRT more efficiently.

B. Static Slack

An important requirement of all schedulable task sets is that for each task τ_i, its WCRT cannot exceed its relative deadline, *i.e.*, $R_i \leq D_i$. We define the difference between the deadline and WCRT as the *static slack* S_i of the task τ_i, *i.e.*,

$$S_i = D_i - R_i \geq 0, \ \forall i. \tag{2}$$

It somehow represents the per-task spare time from a system's perspective and is different from dynamic slack or laxity, as stated in Footnote 1. We further explain why these spare resources are defined as static slack with the following motivational example.

Example II.1. *Consider a sporadic task set τ with three tasks. The task parameters are as shown in Table I. Assuming the (classical) EDF scheduling policy, these tasks' WCRT can be analyzed and are shown in the table. Each task has a slack of either 2 or 3 based on the definition $S_i = D_i - R_i$. Let us consider a modification to the scheduler where each of the released jobs is not* eligible *(i.e., added to the ready queue) immediately, but rather after $S_i = 2$ time units. Correspondingly, each released job instance has a delayed deadline (for priority ordering purposes) that is S_i time units beyond its original deadline. The task set is then scheduled using EDF based on their delayed deadlines (in gray). The resulting schedule is presented in Figure 1. In the figure, the gray cross-hatched represents the idle times caused by such delay/suspension. Notice that although we propose to postpone all deadlines by 2 to 3-time units, all jobs meet their original (non-delayed) deadlines (in black).*

Note that if all jobs are following the original deadlines (in black), the system will still be schedulable under EDF (as the priority order of any pair of jobs remains the same as before). In this situation, one may ignore the delayed deadlines (in gray) and treat the delayed release times (in gray) as the time that a job becomes eligible in Figure 1. The demonstrated schedule remains unchanged and correct.

This example demonstrates the maximum amount of time each job can be delayed from execution without violating its deadline guarantees. Since we can leave the computing resources unused without violating any deadlines, we consider the delay to be the *static slack* (or, in short, slack) time of the task. For the sake of convenience of discussions in later sections, we define a concept now:

Release Slack. Release slack of a job represents the length that the job may not be eligible for execution since its release without affecting system schedulability. A release slack of a task means that each of its released job has at least such amount of release slack. We later prove that upon such (early) suspension, the original deadlines are satisfied for all schedulable task sets.

Section III-A demonstrates how release slack can be safely used by other higher-priority workloads. However, this approach only provides pre-fixed windows of slack, which can be difficult to leverage. In Section III-B, we propose to transfer release slack into a set of Sporadic P-Servers to capture the system slack. Real-time tasks can still be correctly scheduled following the EDF policy, despite the additional workload from the servers. Additionally, since Sporadic P-Servers are sporadic, one can leverage not only the release slack but also the static slack within the entire scheduling window.

Before going into details about the theory and algorithm, we demonstrate such an approach at a high level using the following example.

Example II.2. *Let us consider the same task set τ from Example II.1 with parameters described in Table I. It is possible to calculate the time windows in a hyperperiod where the additional server tasks, each with an execution budget of 1 time unit, can be allowed to execute safely. For the considered task set, the hyperperiod is 30, and the relative deadlines of the five server tasks are 1, 2, 11, 17, and 22, respectively. The server tasks may release sporadically and will be scheduled using EDF policy, just the same as the rest of the tasks. The resulting schedule is presented in Figure 2. The schedule varies from the schedule in Figure 1, yet no task misses any deadline while following EDF. Note that we can only demonstrate the worst-case from schedulability perspectives that all tasks are synchronously released, and the remaining jobs follow a strict periodic release manner. The correctness of other situations will be proven in Section III-B (majorly due to the whole system, including server tasks can be treated as a sporadic task system from a schedulability perspective).*

III. Slack Calculation and Construction

In this section, we propose two methods to capture the available slack of any given task set. Subsection III-A shows that any EDF schedulable system can handle (for free) a *release slack* of up to S_i time units of each job $\tau_{i,j}$. However, this is a static slack at the beginning of jobs' scheduling windows that may not well handle workloads or delays online *at arbitrary time*. To address this, Subsection III-B captures the slack in the form of Sporadic P-Servers, which can be leveraged dynamically during run time in a sporadic manner.

For the sake of convenience of future discussions, we propose two new schedulers for any valid system $\langle \mathcal{T}, \mathsf{EDF} \rangle$:

a) EDF-R Scheduler. For each task $\tau_i \in \mathcal{T}$, we apply an earlier (virtual) deadline as its worst-case response time R_i

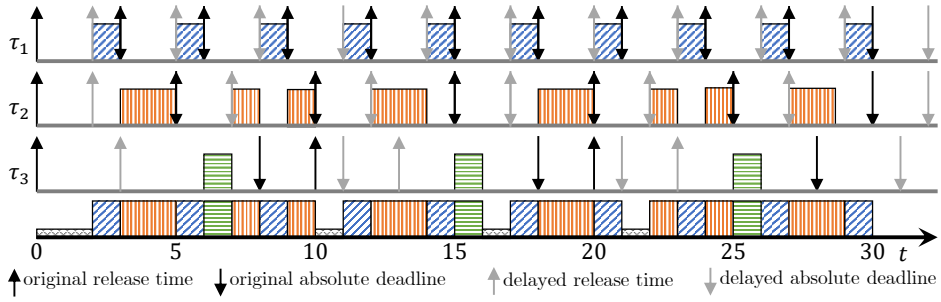

Fig. 1: Schedule of jobs released by the task set in Table I within a hyperperiod, with delayed release times and deadlines following the EDF policy.

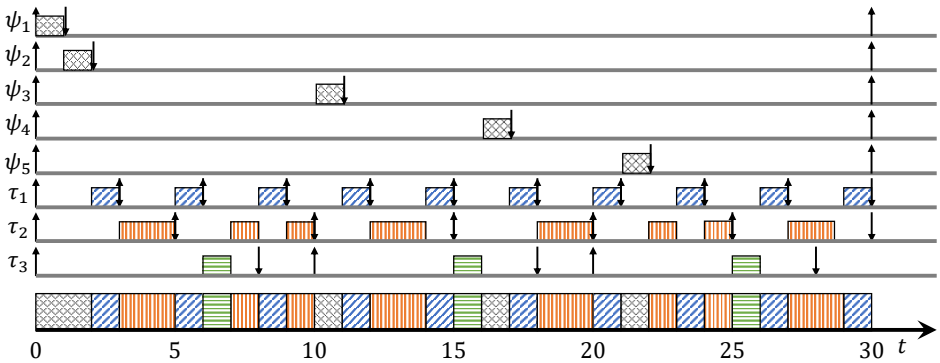

Fig. 2: EDF schedule within a hyperperiod for the set of jobs released by the task set in Table I, in addition to five independent slack (server) tasks (or jobs, as their periods are the same as the hyperperiod) with constrained deadlines at 1, 2, 11, 17, 22, respectively.

($\leq D_i$, due to the feasibility assumption). When the system is scheduled using EDF, workloads in the ready queue are prioritized according to their virtual deadlines. The virtual absolute deadline $\bar{d}_{i,j}$ of a job $\tau_{i,j}$ released by τ_i at $r_{i,j}$ is $\bar{d}_{i,j} = r_{i,j} + R_i$. The EDF with WCRT (EDF-R) scheduler assigns a higher priority to the job with the smaller $\bar{d}_{i,j}$.

b) EDF-D Scheduler. For any job $\tau_{i,j} \in \mathcal{T}$ released at time $r_{i,j}$, we define its eligible time $\bar{r}_{i,j} = r_{i,j} + S_i$, where $S_i = D_i - R_i$ is the static slack of τ_i. The absolute deadline of $\tau_{i,j}$ is $d_{i,j} = r_{i,j} + D_i$. As opposed to classical EDF scheduling where the job is considered eligible for execution at release time $r_{i,j}$, the EDF with Delay (EDF-D) scheduler considers the job eligible for execution at $\bar{r}_{i,j}$. Among all the eligible jobs, EDF-D scheduler executes the one with the smallest $d_{i,j}$.

Note that given any job set, the schedule under EDF-D will be identical to that under EDF-R if for each job $\tau_{i,j}$, the release time is delayed by S_i time units. This way, they share the same deadline settings of all jobs ($S_i + R_i$ for EDF-R and D_i for EDF-D since $\tau_{i,j}$'s release), and also the same eligible window and both follow EDF[2].

Example III.1. *Consider the same task set τ from Example II.1 with parameters described in Table I. Under* EDF-R *scheduler, it is essentially the same as considering a new task set with further reduced deadlines of $D'_1 = 1, D'_2 = 3, D'_3 = 5$, with all other parameters unchanged, scheduled under* EDF*, as depicted in Figure 3. Figure 4 demonstrates how this task set (using original deadlines) would be scheduled under the* EDF-D *scheduler.*

Note that compared to the schedule in Figure 1, although both are enforcing identical delays and following EDF, the deadline settings are not the same: the schedule in Figure 1 following delayed deadlines (gray), while EDF-D follows the (earlier) original deadlines (dark). Later on, we will formally show that such modification of deadlines will not affect schedulability/correctness due to Theorem 1, although the actual schedule may be different.

A. Minimum Static Slack

We say a job is *ineligible* if it is released, but the scheduler chooses not to dispatch it for execution regardless of the priority and processor availability. We now prove that in a feasible system, each job $\tau_{i,j}$ can be ineligible for up to $S_i = D_i - R_i$ time units upon its release without affecting the (preemptive) EDF schedulability of the whole system.

As illustrated in Figure 5, the whole proof consists of three "transition" steps:

(i) Lemma 1 shows that it is safe to "shrink" all deadlines from D_i to R_i, and the WCRT of each task will not increase.

(ii) Lemma 2 shows that after shrinking all the deadlines (to R_i time units apart from release), one can safely "delay" any subset of jobs' release (which is equivalent to suspending the job from the workload perspectives) by exactly S_i time units.

[2]Note that although the actual schedule under EDF-D may be identical as considering each job $\tau_{i,j}$ suspending itself for S_i time units upon release, EDF-D is a system scheduling behavior that enforces such delay while handling jobs/tasks without suspension. This *scheduler* perspective (instead of task perspective) is essential for the proof in Section III-B, which leverages the *optimality* of uniprocessor EDF scheduler (of normal sporadic tasks only).

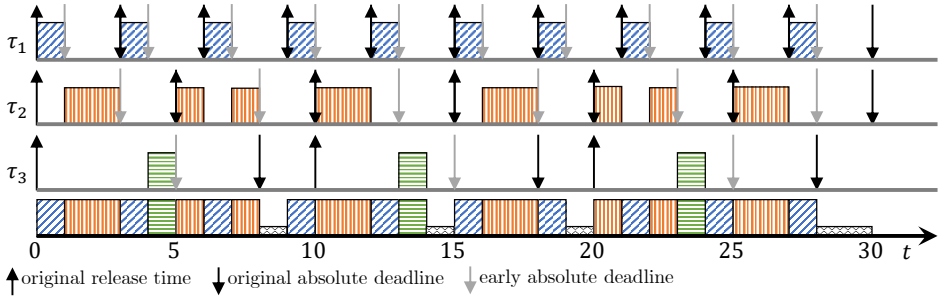

Fig. 3: Schedule of jobs released by the task set in Table I within a hyperperiod, following the EDFR policy.

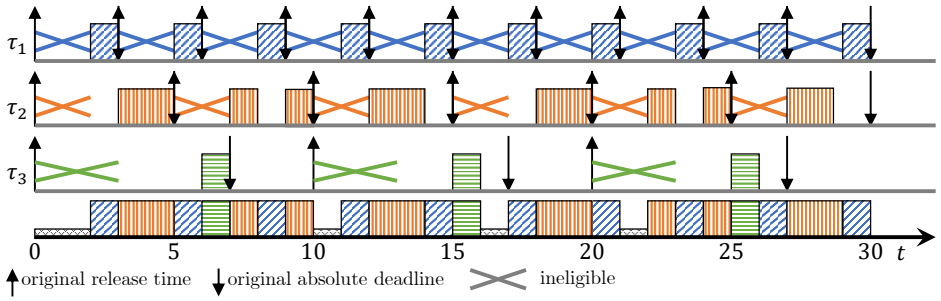

Fig. 4: EDFD Schedule of jobs released by the task set in Table I within a hyperperiod. Note that this schedule is identical to the one in Figure 1, which follows a different deadline setting. It would also be identical to the EDFR schedule if each job is ineligible for S_i time units since its release.

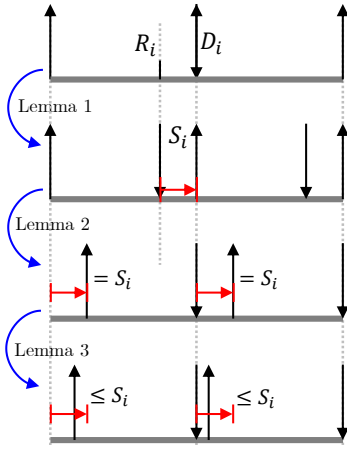

Fig. 5: Illustration of the proof flow of Theorem 2.

The system will still be schedulable under EDF by considering the new deadline(s) as R_i time units after the delayed release, which would be the same as the original deadline upon regular release ($D_i = S_i + R_i$). Equivalently speaking, each task can be ineligible for exactly S_i time units right after its release without affecting system correctness.

(iii) Lemma 3 shows that a release delay of anywhere between 0 and S_i time units will not affect the EDF correctness without modification of the deadlines, as it can be considered as an "early" release comparing to the previous case (with exactly S_i time units of release delay). Equivalently speaking, each task may be ineligible for *up to* S_i time units right after its release without affecting system correctness. Note that the absolute deadline settings for all jobs remain unchanged for this latter transition step, and job priorities are still ordered by those unchanged deadlines.

Lemma 1. *If a task set $\mathcal{T}$ is schedulable by* EDF*, then $\mathcal{T}$ is schedulable by* EDF-R.

Proof. We know that $\mathcal{T}$ is schedulable by EDF. Let $\mathcal{A}$ represent the schedule generated by EDF. The worst-case response time of an arbitrary job $\tau_{i,j}$ in $\mathcal{A}$ is at most R_i. Now consider the new task set $\mathcal{T}'$ with shrunk deadlines $D'_i = R_i$. We know that there exists a correct schedule, $\mathcal{A}$, for this new set. According to Theorem 1, when there is a feasible schedule, following EDF can guarantee all the deadlines to be met (for the new task set, $\mathcal{T}'$). Since $\mathcal{T}'$ has "shrunk" deadlines of $D'_i = R_i$ for any i, EDF for this new task set is exactly EDF-R for the original task set (by the definition of EDF-R scheduler), which must be correct. □

Lemma 2. *If a task set $\mathcal{T}$ is schedulable by* EDF-R*, then $\mathcal{T}$ is schedulable by* EDF *even if each job $\tau_{i,j}$ is* ineligible *for exactly S_i time units right after its release. I.e., $\mathcal{T}$ is schedulable by* EDF-D.

Proof. Note that EDF-R for the original task set is exactly EDF for the new set with virtual deadlines of $D'_i = R_i$. From the fact that synchronous periodic release of all tasks provides the worst-case scenario from EDF schedulability perspectives [4] and EDF already provides a correct schedule for the new task set, we know that any delay of releases to the new set can be considered as a sporadic release pattern, and will be schedulable by EDF[3].

Note that a release delay of exactly S_i time units will cause the delay of the new job's deadline from $D'_i = R_i$ into $R_i + S_i$ time units from its original release, which matches the deadline $D_i = R_i + S_i$ of the original task set. As a result, $\mathcal{T}$ is schedulable by EDF-D. □

So far, we have shown that when the set is scheduled by EDF following original deadlines, each job $\tau_{i,j}$ will receive sufficient execution by its deadline under two situations: (i) immediately ready for execution upon release (and meeting a deadline of R_i), and (ii) be eligible after exactly S_i time units upon release (and meeting a deadline of $D_i = S_i + R_i$). The

[3]This is also the reason why demand bound function based schedulability test is necessary and sufficient for both synchronous periodic task sets and also sporadic task sets under preemptive EDF.

following lemma further relaxes the ability to tolerate such ineligible periods into anywhere between 0 and S_i.

Lemma 3. *If a task set $\mathcal{T}$ is schedulable by* EDF-D*, it is schedulable by* EDF *if each job $\tau_{i,j}$ becomes eligible at anywhere between 0 and S_i time units since its release.*

Proof. We know that $\mathcal{T}$ is schedulable by EDF-D. Let $\mathcal{A}$ represent this schedule. Note that under this schedule, each job $\tau_{i,j}$ is ineligible for exactly S_i time units right after its release.

Now consider a new situation when some jobs become eligible earlier (less than S_i from its release)—consider this as a new task set and treat the eligible time as the release time of each job in the new task set. Compared to $\mathcal{T}$, each job's scheduling (or eligible) window is either enlarged (with a potentially earlier release time) or unchanged (as deadlines remain the same). As a result, $\mathcal{A}$ is still a correct schedule for this new task set.

According to Theorem 1, when there is a feasible schedule, EDF can guarantee all the deadlines to be met for the new task set. Since the absolute deadlines remain unchanged during this transition, which is D_i time units since the release of $\tau_{i,j}$, EDF can guarantee all the deadlines to be met for $\mathcal{T}$ as well, although each job $\tau_{i,j}$ may be ineligible for up to S_i time units since its release. □

Theorem 2. *If a task set $\mathcal{T}$ is schedulable by* EDF*, then under the same deadline setting,* EDF *will guarantee all deadlines being met when each job $\tau_{i,j}$ is "ineligible" for a period of anywhere between 0 and S_i time units since its release, where $S_i = D_i - R_i$ and R_i is the WCRT.*

Proof. It follows immediately from Lemmas 1, 2, and 3. □

In plain words, each job $\tau_{i,j}$'s static slack ($S_i = D_i - R_i$) can be freely "leveraged" at the beginning of its scheduling window without affecting the correctness of the whole system.

B. Static Slack as a function-of-time in Sporadic P-Server Form

One cannot always ensure that the additional suspension or aperiodic workload will align well with the beginning part of scheduling windows of all jobs. In those situations, Theorem 2 can help little in providing the required slack. As a result, and also motivated by Example II.2 and Figure 2, this subsection seeks to formulate the slack in the form of sporadic servers [21], [13]. In addition, motivated by the *optimality* of **P-Fair** [5], our server will be represented as multiple ones, each with an execution budget of 1 time unit. For these reasons, we name it **Sporadic P-Server**.

Sporadic P-Servers are *sporadic* ones such that the starting point of consumption can be freely postponed for each server, providing sufficient freedom to additional workload/suspension requests online. While from a system schedulability point of view, sporadic servers can be treated safely as ordinary sporadic tasks as they were designed carefully to overcome the additional blocking time other servers may impose on lower-priority jobs—see, e.g., Chapter 7.3.3 of [20] for proofs and its schedulability analysis under dynamic priority settings.

Definition 1. *For any valid system $\langle \mathcal{T}, \mathsf{EDF} \rangle$, let us consider a corresponding new system model $(\mathcal{T}^{\mathsf{SP}}, \mathsf{EDF})$ such that*

- *The periodic tasks $\tau_i^{\mathsf{SP}} \in \mathcal{T}^{\mathsf{SP}}$ have no offset and are released Synchronously at time $t = 0$.*
- *Each sporadic constrained deadline task $\tau_i \in \mathcal{T}$ is replaced with a strictly Periodic task τ_i^{SP} in $\mathcal{T}^{\mathsf{SP}}$ whose relative deadline remains unchanged.*

From Lemmas 1 and 2, we know that the system $\langle \mathcal{T}^{\mathsf{SP}}, \mathsf{EDF\text{-}D} \rangle$ has a valid schedule—denoted as $\mathcal{D}$.

For the task set $\mathcal{T}^{\mathsf{SP}}$, we define a static slack budget $\mathcal{B}$:

Definition 2. ***Static slack budget*** *$\mathcal{B}$ is the set of idle instants in the schedule $\mathcal{D}$ generated by $\langle \mathcal{T}^{\mathsf{SP}}, \mathsf{EDF\text{-}D} \rangle$.*

Algorithm 1 presents a pseudocode to calculate the static slack budget $\mathcal{B}$ for $\mathcal{T}^{\mathsf{SP}}$. The budget $\mathcal{B}$ is calculated by emulating the EDF-D scheduler for a given task set (in its synchronized and periodic release form) and storing all the idle instants (where there is no task available for execution) until the hyperperiod.

Specifically, first, a `for`-loop initializes the tasks for execution. Next, a `while`-loop iterates through the hyperperiod H. Within the `while`-loop, tasks that are both eligible and pending execution are added to a priority queue according to their absolute deadlines and index. If the queue is empty at time instant x, then x is idle; otherwise, the task with the earliest deadline is scheduled, and its budget is decremented. Finally, a `for`-loop re-initializes the tasks for future release. The algorithm returns the set of idle points.

Fact 1. *The slack budget $\mathcal{B}$ returned by Algorithm 1 is the set of idle points in $\mathcal{D}$ within a hyperperiod.*

For a valid system $\langle \mathcal{T}^{\mathsf{SP}}, \mathsf{EDF\text{-}D} \rangle$, we define a set of Sporadic P-Server tasks Ψ and their equivalent periodic tasks Ψ^{SP} as follows:

Definition 3. *Ψ is a set of independent Sporadic P-Server tasks with unit execution time and have a period equal to the hyperperiod H of $\mathcal{T}^{\mathsf{SP}}$. Formally,*

$$\Psi = \{(1, H, \delta) \mid \delta \in \mathcal{B}\} \tag{3}$$

Definition 4. *Ψ^{SP} is a set of independent periodic tasks with task parameters identical to their corresponding task in Ψ. Additionally, the periodic tasks $\psi^{\mathsf{SP}} \in \Psi^{\mathsf{SP}}$ have no offset and are released synchronously at time $t = 0$.*

It is trivial that under EDF-D schedule of $\mathcal{T}^{\mathsf{SP}}$, the first $\min_i\{S_i\}$ time units will always be idle. As a result, the following fact holds:

Fact 2. *There are $S_{\min}$ number of Sporadic P-Server tasks with relative deadlines no greater than $S_{\min}$, where*

$$S_{\min} = \min_i\{S_i\}.$$

Algorithm 1: Static slack budget for the task set $\mathcal{T}^{\mathsf{SP}}$

Input : Task set $\mathcal{T}^{\mathsf{SP}}$ with n periodic tasks, task-level slack values $\{S_i \mid \forall \tau_i^{\mathsf{SP}} \in \mathcal{T}^{\mathsf{SP}}\}$
Output: Static slack budget $\mathcal{B}$

```
H ← lcm({T_i | τ_i ∈ T^SP}) ;          // Hyperperiod
B ← φ ;                                // Initialize set
for τ_i^SP ∈ T^SP do
    τ_i^SP.s ← S_i ;                   // Available slack
    τ_i^SP.p ← T_i ;                   // Time to next release
    τ_i^SP.d ← D_i ;                   // Absolute deadline
    τ_i^SP.c ← C_i ;                   // Execution budget
end
x ← 1;
while x ≤ H do
    Q ← φ ;                            // Initialize priority queue
    for τ_i^SP ∈ T^SP do
        if τ_i^SP.s ≤ 0 ∧ τ_i^SP.c > 0 then
            Q.put(⟨τ_i^SP.d, i⟩);
        end
    end
    if Q is not empty then
        τ_i^SP.d, i ← Q.get();   τ_i^SP.c ← τ_i^SP.c − 1;
    else
        B ← B ∪ {x} ;                  // Idle instant
    end
    for τ_i^SP ∈ T^SP do
        τ_i^SP.p ← τ_i^SP.p − 1;   τ_i^SP.s ← τ_i^SP.s − 1;
        if τ_i^SP.p = 0 then
            τ_i^SP.s ← S_i;   τ_i^SP.p ← T_i;
            τ_i^SP.d ← D_i;   τ_i^SP.c ← C_i;
        end
    end
    x ← x + 1;
end
return B
```

Example III.2. *Consider the same task set τ from Example II.1 with parameters described in Table I. Applying Algorithm 1 will yield a slack budget of $B = 1, 2, 11, 17, 22$.*

As a consequence, the system can handle additional Sporadic P-Server tasks of $\psi_1 = (1, 30, 1)$, $\psi_2 = (1, 30, 2)$, $\psi_3 = (1, 30, 11)$, $\psi_4 = (1, 30, 17)$, $\psi_5 = (1, 30, 22)$. Note that $S_{\min} = 2$, and there are two servers with relative deadlines no greater than 2, as illustrated by Figure 2.

Lemma 4. *The task set $\mathcal{T}^{\mathsf{SP}} \cup \Psi^{\mathsf{SP}}$ has a feasible schedule.*

Proof. From Fact 1, $\langle \mathcal{T}^{\mathsf{SP}}, \mathsf{EDF\text{-}D} \rangle$ is valid and has a schedule $\mathcal{D}$. By construction, $\mathcal{B}$ is the set of idle points in schedule $\mathcal{D}$. By definition, Ψ^{SP} has a task with deadline x for each idle instant $x \in \mathcal{B}$ with unit execution time. There is exactly one job $\psi_{i,j}^{\mathsf{SP}}$ released by $\psi_i^{\mathsf{SP}} \in \Psi^{\mathsf{SP}}$. By scheduling the server job $\psi_{i,j}^{\mathsf{SP}}$ within 1 time unit of its corresponding deadline x, $\psi_{i,j}^{\mathsf{SP}}$ meets its deadline x. All jobs released by periodic tasks over an hyperperiod in the system meet their deadlines, and therefore the lemma follows. □

Lemma 5. *The task set $\mathcal{T}^{\mathsf{SP}} \cup \Psi^{\mathsf{SP}}$ is schedulable by* EDF.

Proof. From Lemma 4, $\mathcal{T}^{\mathsf{SP}} \cup \Psi^{\mathsf{SP}}$ has a feasible schedule. Following the optimality of EDF from Theorem 1, the taskset $\mathcal{T}^{\mathsf{SP}} \cup \Psi^{\mathsf{SP}}$ is also schedulable by EDF. □

Theorem 3. *The taskset $\mathcal{T} \cup \Psi$ is schedulable by* EDF.

Proof. From Definition 4, Ψ^{SP} is the periodic equivalent of Ψ. From Definition 1, the task set $\mathcal{T}^{\mathsf{SP}}$ is the periodic equivalent of $\mathcal{T}$. From Lemma 5, the set of periodic tasks $\mathcal{T}^{\mathsf{SP}} \cup \Psi^{\mathsf{SP}}$ is schedulable by EDF. Since a synchronous periodic release represents the worst-case situation from EDF schedulability perspectives, the set of sporadic tasks $\mathcal{T} \cup \Psi$ is schedulable by EDF. □

Computational Complexity. First, we need to calculate the static slack S_i for each task τ_i. This can be computed in pseudo-polynomial time. Next, we need to compute the static slack budget over the hyper-period H. Unfortunately, if the periods of the tasks are co-prime, the hyper-period H could be exponentially large in the worst-case scenario. As a result, the computation of the Sporadic P-Server could take exponential time in the worst case. However, in practice, most workloads have harmonic periods, which means that the periods are co-factors of the largest period. This makes the overall complexity of the algorithm *pseudo-polynomial*.

Remark 3. *Theorem 3 guarantees that Sporadic P-Servers are* safe *to be included in the system, as long as the* replenishment rule *is followed within the* **sporadic server** *'umbrella', such as the simple sporadic server [21] and sporadic background server [13]. Since each Sporadic P-Server task has an execution budget of only 1, and we assume an integer timing model in this paper, the* consumption rules *can be much simpler than existing ones. In Section IV-A, one potential consumption method for aperiodic jobs is discussed in detail.*

Remark 4. *From the way Sporadic P-Server is constructed, we know that, in the worst case, these servers can provide the desired one-unit supply right before their deadline. As a result, the construction approach (Algorithm 1) is* tight*, which means that any attempt to shrink the deadline or enlarge the budget of any server task would lead to an infeasible system.*

System Implementation and Possible Overheads. Although the sporadic P-servers are single-unit ones, their periods are relatively large (hyper-period), and thus, it would not create a huge number of preemptions during each period of any task. In actual implementations, to allow extended executions, one can trace the use of server jobs and their deadlines in the background without creating actual unit-length jobs. This avoids additional context switches, which contribute to the majority of preemption overheads. For example, in Fig. 4, extended execution of any job at Timespots 10, 16, and 21 (in gray) may not cause an additional context switch. Moreover, this concern only applies to the case for Application D, which discussed the usage of Sporadic P-server for overrun in mixed-criticality scheduling (ref. IV-D). However, for Applications B & C (self-suspension) (ref. IV-B and IV-C), the calculated slack can be directly applied to the schedulability test, and during runtime, there is no server actually involved.

IV. Applications

We now discuss how to leverage the proven theoretical results in handling several important application scenarios.

A. Server Consumption and Handling Aperiodic Jobs

Most existing server approaches focus on the direction of each single server, potentially with a large capacity, accommodating multiple aperiodic jobs. By contrast, the servers created by our construction scheme described in Sec. III-B work in the other direction that each aperiodic job receives budgets from multiple server tasks to satisfy its execution requirement. Specifically, Algorithm 1 results in a set of sporadic servers. For each server task, the budget is always the exact 1 time unit and the period of H, which is the hyperperiod of the real-time sporadic tasks for which we have calculated the slacks. Consequently, an aperiodic job with an execution time greater than 1 and a relative deadline at most H—a typical case of aperiodic jobs arriving into the system—would need to consume budgets of multiple servers.

In order to apply multiple such servers to every single aperiodic job in a predictable and systematic manner, two questions must be addressed: (i) **admission control:** whether an arriving aperiodic job can be safely admitted to the system to meet its deadline without affecting the deadline guarantees of existing sporadic tasks and previously admitted aperiodic jobs; (ii) **consumption rule:** if yes for (i), which of the servers should be consumed by this aperiodic job being admitted.

To answer these two questions, we propose an aperiodic job admission algorithm presented as Algorithm 2. Please note that we assume discrete time, i.e., time parameters are all integers. Therefore, the budget of each server invocation must be used up once consumed. This is because of the single-unit budget per server setting by construction, and results in a binary state of each server for any given aperiodic job (use up its budget or do not use its budget at all), assuming that the relative deadline of this aperiodic job at most H. This is also an assumption that Algorithm 2 makes. We will discuss handling aperiodic jobs with relative deadlines exceeding H later by Remark 5.

Since our constructed servers are implemented as sporadic servers, they do not need to be invoked if not to be consumed, but once invoked, the next invocation cannot happen within a server period (H). Therefore, when an aperiodic job is arriving, it is important to know what is the earliest time for the next invocation of each server. For this purpose, an system-wide array $\mathsf{replenish}[\,]$ is maintained by Algorithm 2 whenever an aperiodic job is arriving.

With $\mathsf{replenish}[i]$ initialized as 0 for all i when the system starts (*not* when each time Algorithm 2 is called), at any time instant t, the later one of $\mathsf{replenish}[i]$ and t is the earliest time when server i can make a new invocation. Moreover, if server i is invoked at time t^*, it can only guarantee to provide one unit of budget by time $t^* + \delta_i$, as δ_i is the relative deadline of server i. Therefore, an aperiodic job arriving at time t with an absolute deadline of d can only receive guaranteed budgets from servers such that $\max\{t, \mathsf{replenish}[i]\} + \delta_i \leq d$. Furthermore, note that, in general, the shorter the δ_i, the more "powerful" the server i to provide the budget in an invocation. For example, $\delta_i = 1$ means that the server can provide one unit of budget immediately upon its invocation. Based on this observation, we prioritize consuming the servers with larger δ_i as long as they can provide a guaranteed budget at or before the deadline of the arriving aperiodic job, such that more "powerful" servers remain available to subsequent aperiodic jobs potentially with tighter deadlines. The ideas described in this paragraph yield Algorithm 2. In practice, these servers can be implemented as sporadic tasks, which may be blocked until triggered for release by an incoming aperiodic job. The complexity of identifying the release of the required server would then be comparable to that of a normal sporadic task release, which follows a scheduling policy such as EDF or FIFO.

Algorithm 2: Aperiodic job admission algorithm.

Input : Current time t, where an aperiodic job arrives;
WCET of this job c;
absolute deadline of this job at time d;
number of server tasks $n \leftarrow |\Psi|$;
relative deadlines of server tasks $\{\delta_1, \delta_2, \ldots, \delta_n\}$, in non-decreasing order.
Output: Whether this aperiodic job is admitted to the system, and if yes, the indices ($\mathcal{W}$) of the server tasks this aperiodic job uses.

```
Read replenish[1..n] as rep[1..n];
for i ← n downto 1 do
    if max{t, rep[i]} + δ_i ≤ d and c > 0 then
        W ← W ∪ {i};
        rep[i] ← max{t, rep[i]} + H;
        c ← c − 1;
    end
end
if c = 0 then
    Write rep[1..n] into replenish[1..n];
    return (TRUE, W)
else
    return (FALSE, ∅)
end
```

Remark 5. *Algorithm 2 handles aperiodic jobs with the assumption that their relative deadlines are at most H. Given that H is the hyperperiod of all real-time tasks in the system, we believe this assumption is reasonable for "real-time" aperiodic jobs. Nonetheless, we can still allow aperiodic jobs with relative deadlines exceeding H in the following manner: when such an aperiodic job arrives at time t that has an absolute deadline at $d > t + H$, we do not allow it to use any server until time $d - H$ but just schedule it in the background (i.e., leveraging idle instant without charging servers). At the time $d - H$, we treat it as a new arrival of an aperiodic job with the remaining execution, and Algorithm 2 applies.*

B. Application: Early Self-Suspension

Self-suspension behavior in real-time tasks due to I/O communication or offloading workload to accelerators is modeled using self-suspension task models such as dynamic and seg-

mented self-suspension models. The dynamic self-suspension model is represented by a 4-tuple:

$$\tau_i = (C_i, \sigma_i, D_i, T_i), \tag{4}$$

where σ_i represents the worst-case self-suspension time. The segmented self-suspension model uses an array $(C_i^1, \sigma_i^1, C_i^2, \sigma_i^2, \cdots, C_i^{m_i-1}, \sigma_i^{m_i-1}, C_i^{m_i})$ to denote the alternating m_i computing and $m_i - 1$ suspension segments. The segmented model is a more precise model, in which $\sum_{k=1}^{m_i} C_i^k = C_i$ and $\sum_{k=1}^{m_i-1} \sigma_i^k = \sigma_i^{\mathsf{SP}}$.

Corollary 1. *For a set of n self-suspending sporadic tasks, let R_i represent the worst-case response time of τ_i when scheduled using* EDF *with σ_i (or $\sum_{\forall k} \sigma_i^k$) ignored. Then, the self-suspending tasks are schedulable under* EDF *if suspension behaviors always occur within S_i time units since $\tau_{i,j}$'s release for any job $(\forall i, j)$.*

Proof. This trivially follows from Theorem 2 and the fact that $C_i^{m_i} \leq C_i$: in the worst case when no execution is conducted during the ineligible window, each job can still meet its original deadline by following EDF-D, which prioritizes jobs in the same order as EDF. □

Although this corollary provides suspension allowance simultaneously to all jobs/tasks, suspensions can only occur during the allowed ineligible window at the beginning of each job's scheduling window. This is why we named this subsection "early" self-suspension. A practical application for such a scenario is the I/O read/write operations that are required to be performed before the start of each execution of a task in real-time systems.

C. Application: Arbitrary Self-Suspension

To overcome the limitations in handling early-only self-suspensions, this subsection studies "arbitrary" self-suspension, where the suspension can occur during any execution phase of a task. We focus on the dynamic self-suspension model defined in Equation 4.

First of all, here is some bad news:

Theorem 4. *Even if there is only one self-suspending task in a set $\mathcal{T}$, the system may miss a deadline if the arbitrary self-suspension is cumulatively capped by $S_{\min}$ time units.*

Proof. Consider the task set in Table I with τ_2 being the only self-suspending task with $\sigma_2 = S_{\min} = 2$. See Figure 6 for an illustration of the schedule under EDF (with synchronous release) that the second job of τ_2 misses its deadline at 10. □

This indicates that there is little hope in serving self-suspensions from a task's perspective. It matches existing results in self-suspending task scheduling well and reminds us how "evil" arbitrary suspension can be for real-time schedulability analysis [9]. As a result, we switch our focus to providing guarantees at the hyperperiod level in this subsection and also the following one (which handles task overrun).

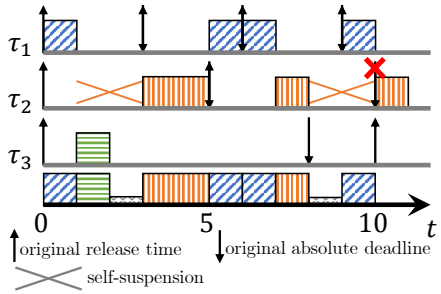

Fig. 6: EDF schedule within [0,11] for the set of jobs released by the task set in Table I, with each job released by τ_2 suspending for $\sigma_2 = 2$ time units arbitrarily and cumulatively. $\tau_{2,2}$ misses its deadline at time 10.

Corollary 2. *Within each hyperperiod, the system can tolerate an accumulated self-suspension of up to $S_{\min} = \min_i\{S_i\}$ time units at any time point.*

Proof. From Fact 2, there are $S_{\min}$ number of unit servers with deadlines shorter than any non-server job (i.e., $S_{\min} \leq S_i < D_i$). From a scheduling perspective, it is always safe to treat suspension as execution [9]. Thus, upon any job's release, we can leverage Algorithm 2 to assign σ_i of these "short-deadline" servers to handle the additional "execution" requirement upon the job's release time—these servers will be scheduled before the job's deadline, and thus the additional "execution" requirement will finish on time. In total, these servers can handle an accumulated self-suspension of up to $S_{\min}$ time units at any time point within a hyperperiod. □

When the accumulated suspension exceeds $S_{\min}$, one may always consider suspension as additional workloads and apply the exact server-charging approach described in Algorithm 2 for more precise handling. If the suspension patterns lie within a certain range, the additional suspension requirements of up to $|\mathcal{B}|$ time units can be tolerated within a hyperperiod. Note that at any time instant, if multiple jobs suspend simultaneously, only one unit of server should be charged.

The provided support to arbitrary suspension is reasoned within a hyperperiod, which seems limited. However, it can be helpful in many situations for modern systems where computations and communications (suspension) are tightly correlated over tasks. For example, consider a component corresponding to a sensor triggered every 10 ms; its suspension is triggered only when any of its intermediate results are needed by another component triggered every 200 ms—which could be at the hyperperiod scale. As a result, although this 10 $ms-$period task would need to be modeled as a self-suspension task, it may suspend very few times within a hyperperiod and can be well-handled by the approach described in this subsection. Unfortunately, the traditional self-suspension task model assumes it suspends every time, and such pessimistic modeling could lead to an infeasible system.

D. Application: Mixed-Criticality Scheduling

Mixed-criticality (MC) design allows components of different levels of importance to be facilitated on a common system.

This can bring benefits in many aspects, such as computation resource utilization, energy consumption, and financial costs. However, it also comes with the potential compromise of providing real-time correctness guarantees. To address this issue, theory, and techniques of real-time MC scheduling have been proposed and investigated.[4]

Assuming a dual-criticality system for simplicity, the most popular MC workload/system model by Steve Vestal [25] allows two WCET estimates, C^L and C^H, to each high-critical task such that $C^L \leq C^H$. In normal circumstances where all tasks finish within its less pessimistic WCET C^L, it is required that all tasks must meet all their deadlines. If any high-critical task overruns its C^L, all high-critical tasks are still required to meet their deadlines as long as they complete within their C^H WCET estimates, whereas low-critical tasks may be sacrificed by dropping or limiting their execution in order to "free up" sufficient computation resource for the overrunning high-critical tasks.

Most existing work on MC scheduling makes such a mode switch at the time when any high-critical task overrunning C^L is detected for the first time and immediately sacrifices the low-critical tasks with the assumption that every single unfinished or subsequent job of every high-critical task will also overrun their C^L estimate. In other words, such an approach makes a dramatic drop in its service guarantees to low-critical tasks even if only a single high-critical task overruns for only one time unit. Such high sensitivity to occasional overrunning can be annoying [12]. Leveraging the results in previous sections, this subsection discusses a scheme, called *Slack-MC*, to potentially handle high-critical tasks overrunning their C^L estimate without triggering mode switch, while maintaining the same schedulability test for existing deadline-driven MC scheduling approaches.

Many dynamic priority approaches use the concept of virtual deadlines to schedule MC tasks in the normal mode, where the virtual deadlines (at or before corresponding actual deadlines) are required to be met for high-critical tasks. With respect to the virtual deadlines, we can apply our proposed methods in Sec. III to calculate the release slacks and construct the set of slack-based sporadic servers. Then, whenever a job of high-critical task overruns, one can treat the overrunning workload (only the part that exceeds C^L) as an aperiodic job that is accommodated by the servers as described in Sec. IV-A, such that system mode switch does not need to be triggered, and if such attempt fails, the mode switch is triggered instead to guarantee MC schedulability still.

Specifically, recall that an array replenish[ ] in initiated as $\vec{0}$ and maintained during runtime by Algorithm 2. When a job of high-critical task τ_k (with virtual deadline at time d_k^V) has executed for C_k^L time units but did not signal its completion at time t^*, we construct an aperiodic job J with arrival time $t = t^*$, WCET $c = C_k^H - C_k^L$, and absolute deadline at $d = d_k^V$, where t, c, d are required input for Algorithm 2. Then, we apply the admission control test, Algorithm 2, to J. If Algorithm 2 returns TRUE, then the replenishment array is updated, no mode switch is triggered, and the overrunning portion is indeed treated as an aperiodic job; if Algorithm 2 returns FALSE, such construction of aperiodic job is voided, and a mode switch is triggered immediately.

Corollary 3. *A mode switch will not happen under* Slack-MC *if the cumulative overrun amount stays no greater than $S_{\min} = \min_i\{S_i\}$ within a hyperperiod. Here, all S_i's are calculated based on the virtual deadlines under normal mode with C_i^L values.*

Proof. From Fact 2, there are $S_{\min}$ number of unit servers with relative deadlines no greater than $S_{\min}$. According to the *Slack-MC* procedures described above, each task will have a slack at least $S_{\min}$ time units earlier than the virtual deadline, if no server is consumed yet. When there are k servers consumed, active jobs can only be delayed for at most k time units. That is, at any time a job is overrunning, if k ($0 \leq k \leq S_{\min}$) servers were consumed (i.e., having a future replenish time), the virtual deadline of this job must be at least $S_{\min} - k$ time units away. In other words, the constructed "aperiodic job" representing the overrun amount must have a deadline of at least $S_{\min} - k$ time units in the future. On the other hand, since only k servers were consumed, there must be at least $S_{\min} - k$ servers with $\delta_i \leq S_{\min} - k$ available (as we consume servers with larger δ_i first). Therefore, Algorithm 2 must return TRUE for this constructed "aperiodic job" if its WCET is at most $\delta_i \leq S_{\min} - k$, which is implied by the corollary statement that cumulative overrun amount stays no greater than $S_{\min} = \min_i\{S_i\}$ as a cumulative overrun amount of k already consumed k servers. □

Remark 6. Slack-MC *can be applied to any deadline-driven MC system and does not affect its schedulability test. On top of that, it can handle limited overrun by* any *task (not necessarily a high-critical one) under the normal mode without triggering a mode switch. While if overrun exceeds the server capacity described in Corollary 3, the system can still safely switch into high mode on time (at or before the virtual deadline of the overrunning job) and provide guarantees to the high-critical tasks, just as in classical Vestal MC systems but without* Slack-MC.

V. Evaluation Results

We have discussed how the proposed static slack calculation and stealing with Sporadic P-Server can be leveraged to handle several application scenarios. We now examine how it performs by comparing it with the state-of-the-art method within each application domain.

Let us first introduce a standard metric for evaluating workload—*utilization*—which is defined as the ratio of execution time and period of a task. For instance, utilization of task τ_i is $u_i = \frac{C_i}{T_i}$, and the utilization of a task set of n tasks is $U = \sum_{i=\{1,\cdots,n\}} u_i$.

[4]In this subsection, we assume that readers are familiar with and have basic knowledge of MC scheduling in real-time systems. If not, we would like to refer readers to [7] for a comprehensive review of this topic, [3] for the specific foundation of handling MC tasks under dynamic priority settings, and [2] for general MC framework.

Workload Generation. For each task set, with a desired utilization $U = \{u \mid 0.1 \leq u \leq 0.9\}$, we use the Uunifast algorithm [6] to generate a set of utilization values for $n = 10$ tasks. The task period T for each task is chosen as the product of four values randomly sampled (with replacement) from $\{2, 3, 5\}$ (such that we may get periods of, e.g., 16, 36, 54, 100, 225, etc.). We use this method as opposed to sampling periods from log-uniform distribution to maintain a manageable hyperperiod. All tasks have implicit deadlines.

A. Response Time of Aperiodic Jobs

For the purpose of evaluating the proposed analysis to accommodate aperiodic jobs through static slacks, the commonly reported schedulability ratio is not relevant. Therefore, we only choose task sets that are schedulable under EDF and evaluate the performance of accommodated aperiodic jobs using 'response time' as a metric for each job. The execution time of the aperiodic jobs is randomly chosen from the interval of 1 to 20 time units. Under each utilization setting, we generate 1000 task sets following the workload generation procedure. For comparison, we choose 'background server [19]', which schedules the jobs whenever the processor is idle and is commonly used to serve aperiodic jobs in real-time scheduling. We compared the proposed servers with background servers instead of well-known sporadic [13], [21] or deferrable [23] servers because laters do not leverage slacks. The results of the evaluation are presented in Figure 7, which shows the response time of the jobs normalized by their hyperperiods. We observe that for lower utilizations $(U < 0.5)$, although the performance of the background server is worse, still comparable to the proposed server. However, as utilization goes beyond 0.5, the background server experiences a significant delay, while the proposed server results in relatively unchanged mean response times for the aperiodic jobs. This behavior can be explained by the reduced availability of idle times for dense workloads. Under such sparse idle time availability, the proposed server allows for earlier than later consumption of these idle instants.

Comparision with CASH [8] and GRUB [18]. *Experimental setup.* the experiments are conducted on 5000 task sets where each task set consists of 5 regular hard real-time tasks and aperiodic workload with an execution requirement of $C = \text{Uniform}(1, (1-U) * H/2)$, where H is the hyper-period of the task set. To compare the performance of our algorithm with CASH and GRUB, we run a simulation for two hyper-periods for the same synthetic workloads for each algorithm, including the vanilla CBS server. *Observations.* Fig. 8 shows the (non-normalized) response times of the jobs of the aperiodic workload. CASH and GRUB are dominated in high workload utilization, which is expected, as these algorithms dynamically utilize the slacks, whereas our algorithm only uses static slacks. However, in low utilization systems, our algorithm performs similarly or better than the CASH and GRUB. It is noteworthy that both CASH and GRUB reclaim resources during runtime, and it is possible to implement the Sporadic P-server with CASH complementarily to utilize both static and dynamic slack.

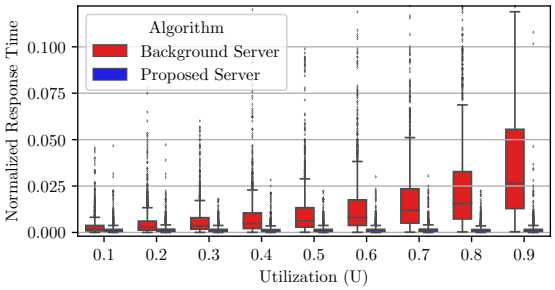

Fig. 7: Comparison showing the distribution of normalized response times (lower being preferred) for aperiodic jobs.

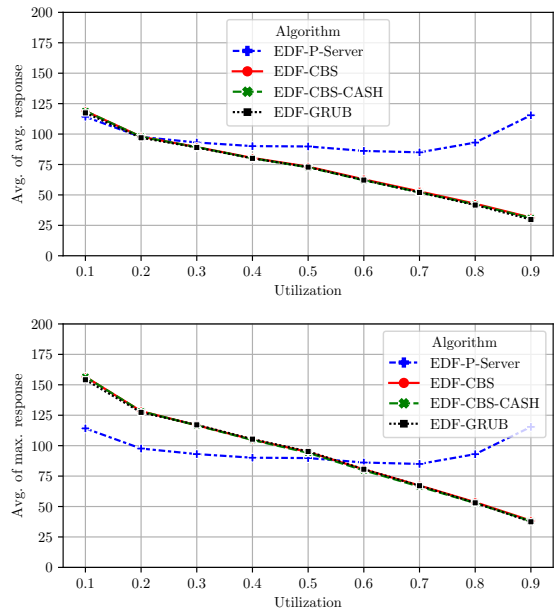

Fig. 8: Comparison of Sporadic P-servers with CASH and GRUB for aperiodic job scheduling showing the average response time (Fig. 8 (Top)), and the maximum response time (Fig. 8 (Bottom)).

B. Early Self-Suspension

We consider the task model described in Equation 4 along with a restriction on the suspension segments to evaluate early self-suspension. The suspension occurs only at the beginning of the job's execution. Note that the parameters—worst-case self-suspension time σ_i and the slack S_i—are not directly related to each other. We only use the S_i parameter to evaluate whether a task set is schedulable by EDF *if* they self-suspend only within the first S_i time units since release. Since there are no existing works with the considered special case scenario, we consider the state-of-the-art EDF scheduling for self-suspending tasks by Günzel *et al.* [14] to evaluate a set of n implicit-deadline periodic tasks with dynamic self-suspension. The workload generation procedure has been discussed, while the self-suspension lengths are sampled from the following ranges under five settings:

- Very short: Log-Uniform$[0.0001(T_i - C_i), 0.1(T_i - C_i)]$
- Short: Uniform $[0.0(T_i - C_i), 0.1(T_i - C_i)]$
- Moderate: Uniform $[0.1(T_i - C_i), 0.3(T_i - C_i)]$
- Large: Uniform $[0.3(T_i - C_i), 0.6(T_i - C_i)]$

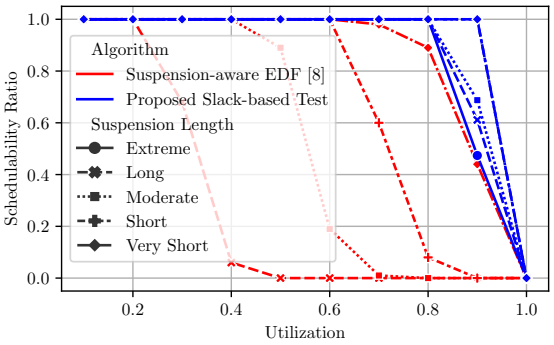

Fig. 9: Schedulability ratio of EDF schedulers for the dynamic self-suspension model under varying system utilization and relative early self-suspension lengths.

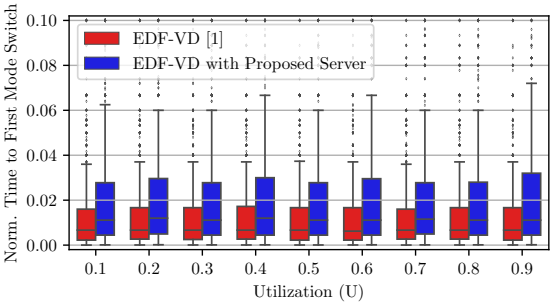

Fig. 10: Comparison showing the distribution of normalized time to first mode switch (higher being preferred) for mixed-criticality task sets.

- Extreme: Uniform $[0.7(T_i - C_i), 1.0(T_i - C_i)]$

The self-suspension generation is chosen to be consistent with the existing literature [14]. We randomly generate 1000 task sets under each configuration and present the results in Figure 9. Since we outperform with 100% schedulability ratio for all suspension lengths considered in [14], we introduce the "extreme suspension length" setting in our evaluations. Our proposed approach is able to schedule almost all task sets up to 0.9 utilization and fails only when availability of slack (or idle points) become too sparse in dense task sets.

C. Mixed Criticality: Time of the First Mode Switch

To evaluate the application to mixed-criticality scheduling, we choose the well-known EDF-VD [3] scheduling algorithm as the baseline. Similar to the previous application, we only choose task sets that are schedulable under EDF-VD. During the scheduling of any given task set, the probability of a high-criticality job exceeding its C^L is set to 10%. The overrun budget $(C^H - C^L)$ is drawn from a log-normal distribution, as opposed to a fixed value. This is to emulate the skewed distribution of the overrun probability. 1000 task sets that meet the workload generation requirements are generated for each configuration. Figure 10 shows the time to first mode switch instant for a task set, normalized by their hyperperiods. The results show a clear delay in the time to the first mode-switch. For brevity and to emphasize the average performance difference, we omit the outliers (can be up to about $1.5H$ time units) from the plot.

VI. Related Works

Aperiodic jobs are typically scheduled with regular hard-deadline periodic/sporadic tasks through server-based resource reservations such as periodic server, sporadic server [16], Constant-bandwidth server (CBS) [1], etc. In addition to server-based approaches, another way to support aperiodic jobs is through static and dynamic slacks of regular jobs. The static slack calculation for fixed-priority scheduling was introduced by the seminal works [10], [11] and has been used in scheduling of hard deadline periodic tasks with hard-deadline aperiodic jobs in fixed-priority jobs [24]. Compared with [24], our proposed method is developed explicitly for EDF scheduling.

Caccamo *et al*. [8] and Lipari and Baruah [18] presented dynamic resource reclamation approaches for CBS-based scheduling, CASH, and GRUB, respectively. CASH [8] scheduled each task in the system using a CBS sever. If any job is finished earlier than the execution budget, the remaining budget with the corresponding deadline is stored in a global queue. Any new job first tries to use the residual execution budget with a deadline equal to or less than its deadline from the global queue. While CASH leverages the unused budgets by other server tasks, GRUB [18] dynamically reclaims the unused *system utilization*. Compared with these algorithms, our proposed algorithm significantly differs from CASH as CASH dynamically reclaimed the unused resource budget considering the variable execution time of each task. Whereas our algorithm only utilizes the static slack for worst-case analysis. Importantly, it is possible to implement both CASH and Sporadic P-servers together to utilize both static and dynamic slacks. Compared with GRUB, our algorithm, and GRUB utilize system-level unused utilization. However, there is a fundamental difference in resource reclamation, such as static (ours) vs. dynamic (GRUB) reclamation. Moreover, both the CASH and GRUB algorithms are developed on CBS and do not use standard EDF algorithms directly for scheduling.

VII. Conclusion and Future Works

This paper builds a simple yet effective relationship between static slack and WCRT under EDF. It further proposes a special Sporadic P-Server to capture static slack precisely. Consumption and replenishment rules are proposed and proven correct. Applications to handling aperiodic jobs, self-suspensions, and task overruns in mixed-criticality systems are handled with extended theory and superior experimental performance demonstrated. The extension to multiprocessor can be challenging. Leveraging WCRT for server construction and consumption rules can be more complicated for the Global-EDF scheduler. Even with partitioned EDF, if we allow aperiodic jobs to migrate, it is not trivial to choose the proper combination of Sporadic P-Servers among multiple processors to serve one job. We leave these as future work.

Acknowledgement

We are grateful to the anonymous reviewers for their valuable feedback and the anonymous shepherd for guiding

us to improve the paper. This research has been supported in part by NSF Award FRR-2246672, CNS-2104181, and startup funding from North Carolina State University.

REFERENCES

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No. 98CB36279)*, pages 4–13. IEEE, 1998.

[2] A. A. Arafat, S. Vaidhun, L. Liu, K. Yang, and Z. Guo. Compositional mixed-criticality systems with multiple executions and resource-budgets model. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 67–79. IEEE, 2023.

[3] S. Baruah, V. Bonifaci, G. D'angelo, H. Li, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie. Preemptive uniprocessor scheduling of mixed-criticality sporadic task systems. *Journal of the ACM (JACM)*, 62(2):14, 2015.

[4] S. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings 11th Real-Time Systems Symposium (RTSS)*, pages 182–190, 1990.

[5] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[6] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-time systems*, 30(1-2):129–154, 2005.

[7] A. Burns and R. I. Davis. A survey of research into mixed criticality systems. *ACM Computing Surveys (CSUR)*, 50(6):82, 2017.

[8] M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proceedings 21st IEEE Real-Time Systems Symposium*, pages 295–304. IEEE, 2000.

[9] J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, N. Audsley, et al. Many suspensions, many problems: a review of self-suspending tasks in real-time systems. *Real-Time Systems*, 55(1):144–207, 2019.

[10] H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on software engineering*, 15(10):1261, 1989.

[11] R. I. Davis, K. W. Tindell, and A. Burns. Scheduling slack time in fixed priority pre-emptive systems. In *1993 Proceedings Real-Time Systems Symposium*, pages 222–231. IEEE, 1993.

[12] R. Ernst and M. Di Natale. Mixed criticality systems—a history of misconceptions? *IEEE Design Test*, 33(5):65–74, 2016.

[13] T. M. Ghazalie and T. P. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems*, 9(1):31–67, 1995.

[14] M. Günzel, G. von der Brüggen, and J.-J. Chen. Suspension-aware earliest-deadline-first scheduling analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):4205–4216, 2020.

[15] W. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21(1):177–185, 1974.

[16] H. Kaneko, J. A. Stankovic, S. Sen, and K. Ramamritham. Integrated scheduling of multimedia and hard real-time tasks. In *17th IEEE Real-Time Systems Symposium*, pages 206–217. IEEE, 1996.

[17] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Proceedings. Real-Time Systems Symposium (RTSS)*, pages 166–171, 1989.

[18] G. Lipari and S. Baruah. Greedy reclamation of unused bandwidth in constant-bandwidth servers. In *Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS 2000*, pages 193–200. IEEE, 2000.

[19] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[20] J. W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, 1st edition, 2000.

[21] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.

[22] M. Spuri. Analysis of deadline scheduled real-time systems. *[Research Report] RR-2772*, 1996. inria-00073920.

[23] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44(1):73–91, 1995.

[24] S. R. Thuel and J. P. Lehoczky. Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing. In *RTSS*, pages 22–33, 1994.

[25] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *the 28th IEEE Real-Time Systems Symposium (RTSS'07)*, 2007.

[26] F. Zhang and A. Burns. Schedulability analysis for real-time systems with EDF scheduling. *IEEE Transactions on Computers*, 58(9):1250–1258, 2009.