2018 IEEE Real-Time and Embedded Technology and Applications Symposium

Physical-State-Aware Dynamic Slack Management for Mixed-Criticality Systems

Hoon Sung Chwa and Kang G. Shin
Electrical Engineering and Computer Science
The University of Michigan, Ann Arbor, Michigan, U.S.A.
{hchwa, kgshin}@umich.edu

Hyeongboo Baek and Jinkyu Lee
Department of Computer Science and Engineering
Sungkyunkwan University (SKKU), Republic of Korea
{hbbaek, jinkyu.lee}@skku.edu

***Abstract*—Safety-critical cyber-physical systems like autonomous cars require not only different levels of assurance, but also close interactions with dynamically-changing physical environments. While the former has been studied extensively by exploiting the notion of mixed-criticality (MC) systems, the latter has not, especially in conjunction with MC systems. To fill this important gap, we conduct an in-depth case study, demonstrating the importance of capturing current *physical states*, and introduce the problem of achieving efficient utilization of computing resources under varying physical states in MC systems. To solve this problem, we first develop a *physical-state-aware MC task model*, which is a generalization of the existing basic MC task model. We then propose new *slack* concepts tailored to the new task model, which enable efficient utilization of computing resources for MC systems. Finally, we develop a physical-state-aware dynamic slack management framework and demonstrate how to utilize the new MC task model and slack concepts towards efficient system utilization. We show, via a case study and in-depth evaluation, that the proposed framework makes 20x less low-criticality jobs dropped over a popular MC scheduling algorithm without compromising the MC-schedulability requirements.**

I. Introduction

Safety-critical cyber-physical systems are increasingly realized as *mixed-criticality* (MC) systems, where multiple functions with different safety-criticality levels are integrated on a shared computing platform. At the same time, cyber-physical systems are becoming highly dynamic and closely interacting with their physical environments.

Advanced driver assistance system (ADAS) and autonomous cars are prototypical examples that combine both mixed-criticality and dependency on physical environments. An ADAS implementation requires to meet the safety standard, ISO 26262, specifying different levels of safety assurance, which corresponds to the MC requirement. Our case study of adaptive cruise control (ACC), a typical ADAS feature, reveals that the computation time of control signals is strongly correlated with a *physical state* of the car, such as the distance to the car ahead and its relative speed. We have also found it taking less computation time to update a control signal if the preceding car is far away and the host/follower car travels at the speed set by the driver (i.e., in a steady state). On the other hand, the amount of computation is increased by up to 20x when the distance to the preceding car is too close to the follower car. Moreover, the actual execution time is found to change widely over time within the same physical state and show a considerable difference in the WCET (worst-case execution time) estimates (to be detailed in Section II). These distinct features pose significant challenges in achieving efficient resource utilization under varying physical states while assuring the safety associated with tasks of different criticality levels in MC systems.

Vestal [1] proposed the classic MC task model for 2 criticality levels, where a high-criticality (HC) task has 2 WCET estimates with different levels of confidence (criticality). Based on the MC task model, a vast amount of work has been done on MC scheduling to ensure the satisfaction of MC tasks' deadlines. A MC system is usually seen to be in two different execution modes at runtime: LC (low-criticality) and HC modes. The system starts in LC mode during which all tasks are assigned resources based on their low-confidence WCET estimates and scheduled together. Once any HC task executes more than its low-confidence WCET estimate, the system switches to HC mode during which only HC tasks can execute up to their high-confidence WCET estimates and no deadlines of LC tasks are required to be met. Although these approaches ensure the execution of critical tasks, they do not capture dynamically changing resource demands as the physical state changes. The underlying assumption of the classic MC task model is that the two WCET estimates do *not* change during runtime. Those WCET estimates are derived independently of physical states under pessimistic assumptions. Thus, existing approaches under the classic MC task model use *static* resource allocation without considering the underlying physical state. As a result, they under-utilize resources by triggering a mode-switch unnecessarily and penalizing all LC tasks indiscriminately.

Some recent studies [2–6] support dynamic resource allocation exploiting slack (spare capacity) in order to reduce the pessimism of the traditional static approaches, but all of them have at least one of the following limitations. First, they consider only a statically available slack by using an offline schedulability analysis, although the amount of slack can vary with time. Second, they assume that the WCET budget is statically assigned. They cannot capture WCET variations with the physical state when calculating a dynamic slack. Third, they do not provide any slack tailored to MC systems, failing to adaptively switch to different modes; for example, there is no dynamic slack available for use by LC tasks in HC mode. Such limitations make it impossible/difficult to take varying physical states into account when dealing with a dynamic slack, leading to dropping LC tasks unnecessarily.

Our approach. We propose a physical-state-aware dynamic slack management framework for MC systems. It will be able to capture varying resource demands depending on the

978-1-5386-5295-4/18/$31.00 ©2018 IEEE
DOI 10.1109/RTAS.2018.00023

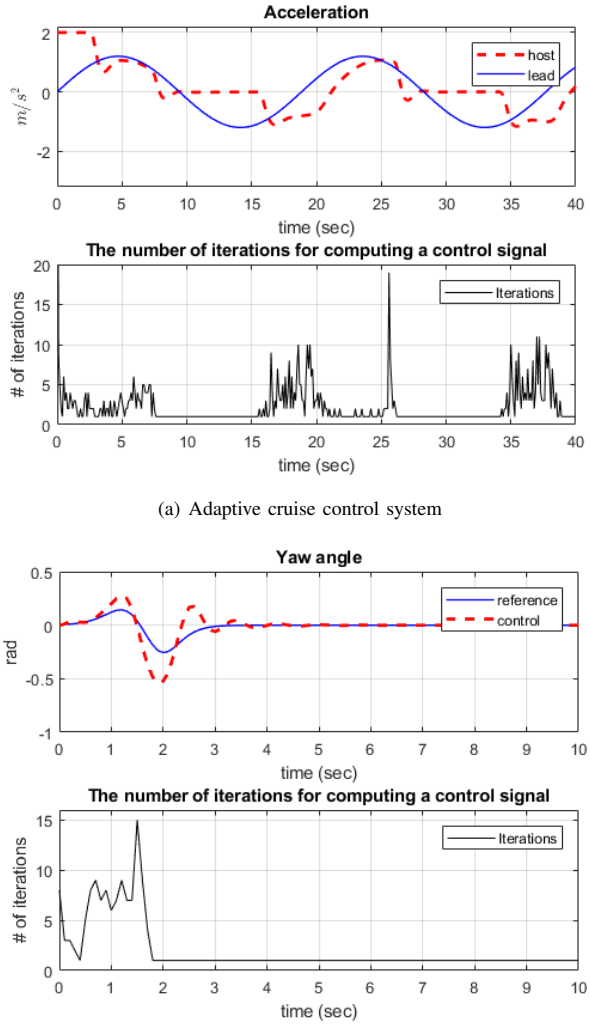
physical state and calculate the corresponding available slack at runtime. By utilizing a dynamic slack, we can schedule more LC tasks while still meeting the MC requirement of the entire system. The main challenge is then how to calculate a tight lower bound on dynamic slack under varying physical states and how to schedule the slack so as to minimize the number of LC job drops without compromising MC-schedulability. To meet this challenge, we first propose a physical-state-aware MC task model for characterizing tasks that have dynamically changing execution times depending on the external physical events. We extend the static WCET estimates in the classic MC task model [1] to vary with the physical state.

Second, we extend the notion of slack capturing the MC requirement, where different sets of deadlines should be met at different criticality levels, with a novel notion of *LC- and HC-mode slacks*. The LC-mode slack can capture the amount of surplus time available in LC mode, while guaranteeing all tasks to execute for their low-confidence WCETs by their deadlines. Likewise, the HC-mode slack can express the amount of surplus time available in HC mode, while guaranteeing HC tasks to execute for up to their high-confidence WCETs by their deadlines. These new slack concepts tailored to MC systems enable utilization of available resources depending on the criticality mode.

Third, building upon the new MC task model and the slack concepts, we develop a physical-state-aware dynamic slack management framework. It assigns a resource budget to each task dynamically at runtime by using the WCET estimate corresponding to the current physical state and updates LC- and HC-mode slacks accordingly upon change of the physical state at runtime. Within the allocated resource budget, tasks are scheduled according to the well-known Earliest Deadline First with Virtual Deadlines (EDF-VD) MC scheduling policy [7]. If any task tries to execute beyond its resource budget, our framework allocates additional resources to it according to the LC-mode (HC-mode) slack in LC mode (HC mode). In particular, the LC-mode slack can be used for a HC task to execute beyond its low-confidence WCET without triggering a mode-switch. After a mode-switch is made, the HC-mode slack can be used for an LC task to execute without compromising the execution of HC tasks. We also propose a slack-based mode-switch mechanism based on the observation that the amount of available slack can be used to determine a mode-switch. The proposed framework has been evaluated using both an ADAS case study and an extensive simulation of synthetic task sets, showing a 20x reduction of the number of dropped LC jobs over the case of using EDF-VD alone. Our framework is shown to become much more effective in reducing the number of LC job drops for a task set with a larger number of tasks, each of which has lower utilization, than the one with a smaller number of tasks, each of which has higher utilization.

This paper makes the following main contributions:

- An insightful case study that shows dynamic execution behavior depending on the physical state and the importance of accounting for such dynamics in MC scheduling (Section II);
- A new MC task model that enables the characterization of tasks with dynamically changing execution behaviors according to their physical states (Section III);
- A new slack concept tailored to MC scheduling for efficient resource utilization (Section IV);
- A runtime dynamic slack management framework that enables adaptive resource allocation under varying physical states and efficient slack usage while guaranteeing MC-schedulability (Section V); and
- Demonstration of the effectiveness of the proposed framework, via a case study and in-depth evaluation, minimizing service degradation in low-criticality tasks (Section VI).



(a) Adaptive cruise control system

(b) Autonomous vechicle steering system

Fig. 1. Motivational case study: an ADAS system

II. Motivational Case Study and Goal

We first present a motivational case study of ADAS to demonstrate the dynamic execution behavior of each component according to the physical state of its corresponding controlled plant and also introduce other applications. We will

then make an important observation from the case study and state the goal of our new MC task model and dynamic resource management thereof.

A case study. An ADAS system consists of two components — adaptive cruise control (ACC) and autonomous vehicle steering (AVS) — which run on a shared computing platform. The ACC periodically adjusts the vehicle speed to maintain a safe distance from the front vehicle, and the AVS affects lateral and steering maneuvers of the vehicle to avoid any collision.

For experimental evaluation, we implement ACC and AVS components in Matlab using model predictive control (MPC) [8], which has recently attracted considerable interest from the automotive domain.[1] We set the driver's desired speed to $30m/s$ while varying the acceleration of a preceding vehicle and the reference trajectory including occasional double lane change maneuvers. We set the sampling period of each controller to 0.1 second and simulate the system for 40 seconds.

Figs. 1(a) and (b) show the simulation results. The upper part of each figure shows the control output along with the reference trajectory, while the bottom part shows the amount of computation required to compute each control output at every sampling interval. Note that MPC operates by repeatedly solving the *iterative* online optimization problem on a finite prediction horizon initialized with the current physical state. The constraints on control inputs and states may vary with the current physical state, yielding a different number of iterations taken to solve the optimization problem at each sampling interval. Also, note that the number of iterations for optimization can be translated into the amount of execution time required for computation.

As shown in the figure, the number of iterations performed for each component varies dynamically with its control output and state. For example, in the case of ACC, one iteration was enough to compute a control output when the preceding vehicle was far away and the host vehicle traveled at a constant desired speed (i.e., in a steady interval from 9 to 15 seconds). In contrast, the number of iterations was increased by up to 20x when the distance to the preceding vehicle became too close, needing to maintain a safe distance (i.e., in a transient state from 15 to 28 seconds). In addition, the change in the number of iterations in the transient interval was highly dynamic over a wide range. Note that a similar behavior was also exhibited in the case of AVS.

Let us provide some high-level ideas of how to derive different WCET values according to the physical state, although it is beyond the scope of this paper. In case of MPC, solving the quadratic programming (QP) problem is the most dominant computation requirement. According to [11], the computation time is highly dependent on the number of *active* constraints in QP. So, in our simulation to be presented later in Section VI, we divided the physical state of ACC or AVS into two, depending on whether there is an active constraint or not. For example, ACC has no active constraint when the actual distance from the front vehicle is larger than the safety distance on the prediction horizon. We can then calculate WCET bounds for two physical states. Vestal [1] already provided a good way of determining low- and high-confidence WCET estimates for each physical state. A low-confidence WCET estimate can be obtained by the worst-case time observed during extensive tests, while a high-confidence WCET estimate can be obtained by some code flow analysis and worst-case instruction cycle counting under pessimistic assumptions.

[1]See [9, 10] for the relevant vehicle dynamics and implementation details.

Other applications. There exist other applications — engine control and vision-based object recognition — that show the dynamic execution behavior depending on the physical state. According to [12–16], the typical approach for engine control used in automotive industry is to select different functions to be executed depending on the speed of the engine crankshaft's rotation. The WCET estimates of an engine control task can thus be expressed as a step function of the rotational speed. According to [17], the execution time of a vision-based object recognition task is categorized into two, depending on whether or not there are a large number of objects in the camera's field-of-view.

In addition to MPC, we can also consider a *switched control system* that consists of several different controllers and switches among them to achieve better control performance under varying physical states. We can derive WCET estimates for each individual controller and use different WCET estimates whenever the controller changes. Such a control design is well-known and has numerous applications in the control of mechanical systems, process control, automotive control, power systems, aircraft and traffic control, and so on [18].

Motivation and goal. Without considering such dynamic execution behavior that depends on the physical state, we may severely under-utilize computing resources or even substantially degrade the service of low-criticality tasks. Suppose both ACC and AVS tasks can be modeled as the classic MC task model, under which each task has *fixed* WCET estimates (i.e., average- and worst-case execution times during the *entire* service time) and is always assigned its resource based on the WCET estimates. Then, a considerable amount of computing resources will remain unused throughout all the sampling intervals where the iteration count is below the average value. If those unused resources could be reclaimed and utilized for other tasks properly, we can minimize degradation in servicing low-criticality tasks while guaranteeing MC-schedulability. The main challenge arises from the fact that we cannot predict the varying physical states *a priori*. This requires a new MC task model that can express WCET estimates as a function of the physical state, as well as a new notion of *slacks* tailored to MC scheduling for a better use. Building upon the new task model and slack concepts, we want to develop an online dynamic slack management framework that assigns resources dynamically under varying the physical state, reclaims unused resources efficiently, and utilizes them so as to minimize service degradation in low-criticality tasks without compromising MC-schedulability.

III. New MC Task Model

This section introduces a new physical-state-aware MC task model, which is a generalization of the existing MC task model [1].

In this new MC task model, each task $\tau_i \in \tau$ can be specified as $\tau_i = (T_i, C_i, D_i, L_i)$, where T_i is the minimum separation between successive job releases, C_i is a list of WCET values, D_i ($\leq T_i$) and L_i denote the relative deadline and the criticality level, respectively. We assume that every task τ_i has either low (denote by LC) or high criticality (denoted by HC). Let τ^L and τ^H denote a set of LC and HC tasks in τ, respectively.

C_i is expressed by $\{C_i^L(s_i), C_i^H(s_i)\}$, where $C_i^L(s_i)$ and $C_i^H(s_i)$ denote LC and HC WCET of τ_i at its physical state s_i, respectively. We assume that $C_i^L(s_i) \leq C_i^H(s_i)$ for every HC task, and $C_i^L(s_i) = C_i^H(s_i)$ for every LC task. Jobs of τ_i are released with a minimum separation of T_i time units; each job $J_{i,j}$ (the jth job of τ_i) can execute for no more than $C_i^H(s_{i,j})$ time units, and should finish its execution within D_i time units since its release, where $s_{i,j}$ denotes the physical state when job $J_{i,j}$ of τ_i is released. Let $r_{i,j}$ and $d_{i,j}$ denote the release time and deadline of $J_{i,j}$. For each job $J_{i,j}$, its first execution up to $C_i^L(s_{i,j})$ is called *LC-part execution*, while its next execution up to $C_i^H(s_{i,j}) - C_i^L(s_{i,j})$, followed by its LC-part execution amounting to $C_i^L(s_{i,j})$, is called *HC-part execution*.

For the ease of presentation, we introduce the following notation:

$$C_i^{L,\max} = \max_{s_i} C_i^L(s_i), \quad C_i^{H,\max} = \max_{s_i} C_i^H(s_i). \tag{1}$$

We express the utilization of each task with given current physical state and criticality as:

$$u_i^L(s_i) = \frac{C_i^L(s_i)}{T_i}, \quad u_i^H(s_i) = \frac{C_i^H(s_i)}{T_i}. \tag{2}$$

Also, for notational convenience, we define the following utilization:

$$u_i^{x,\max} = \max_{s_i} u_i^x(s_i), \quad U_y^x = \sum_{\tau_i \in y} u_i^{x,\max}, \tag{3}$$

where $x \in \{L, H\}$ and $y \in \{\tau^L, \tau^H\}$.

In this paper, we focus on implicit-deadline task systems in which relative deadline D_i is equal to T_i for every task $\tau_i \in \tau$, and consider the problem of scheduling n such tasks $\tau = \{\tau_1, \ldots, \tau_n\}$ on a uniprocessor platform.

Like the Vestal's task model [1], we can then define schedulability under the proposed MC task model as follows.

Definition 1 (MC-Schedulable): A system τ in the physical-state-aware task model is defined to be *MC-Schedulable* by a scheduling algorithm if the following two conditions hold:

1) If there does not exist any job (invoked by a task in τ) that executes for more than its LC WCET, every job (invoked by a task in τ) finishes its execution (for at most its LC WCET) before its deadline.
2) If such a job exists, every HC job (invoked by a task in τ^H) finishes its execution (for at most its HC WCET) before its deadline.

The physical-state-aware MC task model is more general than the Vestal's task model, as stated in the following lemma.

Lemma 1: The physical-state-aware MC task model subsumes the Vestal's MC task model [1].

Proof: If we assign $C_i^L(s_i) = C_i^{L,\max}$ and $C_i^H(s_i) = C_i^{H,\max}$ for every pair of (τ_i, s_i), the physical-state-aware MC task model becomes the usual MC task model. ■

Since the proposed physical-state-aware MC task model can express multiple WCETs according to different physical states in conjunction with different criticality levels, the model together with the concepts to be introduced in Section IV provides an interface for efficient resource reservation, which will be realized in Section V.

IV. New Slack Concept

Since real-time systems provide timing guarantees based on the worst-case behaviors, they severely under-utilize computing resources. The notion of slack has been widely used in non-MC task models [19, 20] to utilize reserved-but-unused (surplus) resources [21–23]. In this section, we generalize the concept of slack in non-MC task models, to the proposed physical-state-aware MC task model (which is also applicable to the Vestal's MC task model [1]).

As shown in Definition 1, the amount of resources reserved/guaranteed for each job depends on the system criticality levels—whether 1) or 2) in the definition. This necessitates the notion of different slacks for different system criticality levels (two levels in this paper).

Definition 2 (LC-mode slack): Suppose that under a work-conserving scheduling algorithm, every job in $[t_1,\ t_2)$ performs its LC-part execution (for at most its LC WCET) and does not perform its HC-part execution, and there is no job deadline miss in $[t_1,\ t_2)$. In this case, we define a LC-mode slack in $[t_1,\ t_2)$ under the scheduling algorithm, denoted by $S_{LC}(t_1, t_2)$, as the amount of idle time in $[t_1,\ t_2)$.

Definition 3 (HC-mode slack): Suppose that under a work-conserving scheduling algorithm, every LC and HC job in $[t_1,\ t_2)$ executes for zero and at most its HC WCET time units, respectively, and there is no HC job deadline miss in $[t_1,\ t_2)$. In this case, we define an HC-mode slack in $[t_1,\ t_2)$ under the scheduling algorithm, denoted by $S_{HC}(t_1, t_2)$, as the amount of idle time in $[t_1,\ t_2)$.

Note that Definitions 2 and 3 are new concepts not only for the proposed physical-state-aware MC task model, but also for the classic MC task model [1].

Once we calculate LC- and HC-mode slacks, we can utilize those slacks for efficient system utilization. That is, the LC-mode slack implies the amount of HC-part execution of an HC job without triggering a mode-switch, while the HC-mode one does the amount of LC-part execution of an LC job without compromising other HC jobs' execution, as stated in the following lemmas.

Lemma 2: Let $J_{i,j}$ denote an HC job, which is active at t_1 and whose deadline is t_2 ($> t_1$). Suppose that under a work-conserving scheduling algorithm, every job in $[t_1,\ t_2)$ performs its LC-part execution (for at most its LC WCET) and does not

perform its HC-part execution, and there is no job deadline miss in $[t_1,\ t_2)$; in addition, $S_{LC}(t_1, t_2) \geq 0$ holds. Then, there is a schedule that an increase of the execution of $J_{i,j}$ up to $C_i^L(s_{i,j}) + S_{LC}(t_1, t_2)$ does not cause any other jobs to miss their deadlines within $[t_1,\ t_2)$.

Proof: If we assign the lowest priority to the HC-part execution of $J_{i,j}$, there is no schedule change of other jobs in $[t_1,\ t_2)$, implying that none of other jobs misses its deadline in $[t_1,\ t_2)$. Note that it is possible to design different scheduling policies to avoid deadline misses, by considering the property of the target scheduling algorithm. ■

Lemma 3: Let $J_{i,j}$ denote an LC job, which is active at t_1 and whose deadline is t_2 ($> t_1$). Suppose that under a work-conserving scheduling algorithm, every LC and HC job in $[t_1,\ t_2)$ executes for zero and at most its HC WCET time units, respectively, and there is no HC job deadline miss in $[t_1,\ t_2)$; in addition, $S_{HC}(t_1, t_2) \geq 0$ holds. Then, there is a schedule that an increase of execution of $J_{i,j}$ up to $S_{HC}(t_1, t_2)$ does not cause any other HC jobs to miss their deadlines in $[t_1,\ t_2)$.

Proof: The proof is similar to that of Lemma 2; assigning the lowest priority to the LC-part execution of $J_{i,j}$ does not change the schedule of other HC jobs in $[t_1,\ t_2)$. ■

An example of illustrating the usage of Lemmas 2 and 3 is provided in Appendix A (see Fig. 3) of the supplement file [24].

While Definitions 2 and 3, and Lemmas 2 and 3 offer concepts and principles necessary for utilizing *observed* slack values for the MC system under any target scheduling algorithm, they do not specify how to calculate the slacks no later than the beginning of the interval of interest. In Section V, we will show how to calculate lower-bounds of slacks under a given target scheduling algorithm *at* the beginning of the interval of interest. This calculation enables to i) delay a mode-switch by executing HC jobs' HC-part execution without triggering a mode-switch (by Lemma 2) and ii) reduce the number of LC job drops by executing LC jobs' LC-part execution without compromising other HC jobs' execution (by Lemma 3), both to be discussed in Section V.

V. Physical-State-Aware Dynamic Slack Management

Sections III and IV proposed two tools that can be utilized for efficient resource usage by considering varying physical states and MC systems. Based on the two tools, we will propose a physical-state-aware dynamic slack management framework, which not only captures how slack values dynamically change at runtime, but also achieves high resource-efficiency by utilizing the runtime slack dynamics analysis. To this end, we present when and how to update/calculate slacks at runtime and how to use the calculated slacks for efficient resource utilization.

A. Target Scheduling Algorithm

Here we focus on EDF-VD [7] scheduling. Basically, all jobs are scheduled by the Earliest Deadline First (EDF) policy [19]. Under EDF-VD, HC jobs are assigned virtual deadlines shorter than their original deadlines in LC-mode. These virtual deadlines ensure that when a mode-switch occurs, there is a sufficient time for HC jobs to complete any additional execution before their original deadlines. Let $VD_i = x \cdot T_i$, where $x \in [0, 1]$, denote the virtual deadline for each task $\tau_i \in \tau^H$. Then, EDF-VD operates as follows:

- In LC mode, a job of HC task $\tau_i \in \tau^H$ is assigned a scheduling deadline equal to its release time plus its virtual deadline VD_i, while a job of LC task $\tau_j \in \tau^L$ is assigned a scheduling deadline equal to its release time plus its original deadline D_j; and
- In HC mode, the scheduling deadline of an HC job of $\tau_i \in \tau^H$ that is active at a mode-switch is changed to its release time plus its original deadline D_i. A future HC job of τ_i is assigned a scheduling deadline equal to its release time plus its original deadline D_i. LC jobs will not receive any execution.

We can use the schedulability test [7] for EDF-VD and the classic MC task model to find a feasible virtual deadline assignment for EDF-VD and the physical-state-aware MC task model, as stated in the following lemma.

Lemma 4: A task set τ is MC-schedulable by EDF-VD on a uniprocessor platform if there exists a virtual deadline scaling parameter x such that

$$\frac{U_{\tau_H}^L}{1.0 - U_{\tau_L}^L} \leq x \leq \frac{1.0 - U_{\tau_H}^H}{U_{\tau_L}^L}. \tag{4}$$

Proof: By definition, $U_{\tau_L}^L$, $U_{\tau_H}^L$, and $U_{\tau_H}^H$ are upper-bounds on $\sum_{\tau_i \in \tau^L} u_i^L(s_i)$, $\sum_{\tau_i \in \tau^H} u_i^L(s_i)$, and $\sum_{\tau_i \in \tau^H} u_i^H(s_i)$, respectively. Then, using Theorems 1 and 2 in [7], it trivially holds that τ is MC-Schedulable by EDF-VD on a uniprocessor platform with the virtual deadline scaling parameter x, if Eq. (4) holds, thus proving the lemma. ■

Note that Lemma 4 assumes that every job of each task τ_i is assigned its LC and HC WCET as $C_i^{L,\max}$ and $C_i^{H,\max}$, respectively, regardless of the physical state. We can make a dynamic virtual deadline assignment while considering the physical state, but it is out of scope of this paper. Instead, we focus on dynamic resource allocation and slack management while considering the physical state for a given feasible virtual deadline assignment. We assume that any value of the virtual deadline scaling parameter x satisfying Eq. (4) can be used.

B. Runtime Behavior of Dynamic Slack Management

Our physical-state-aware dynamic slack management framework employs the prioritization policy of EDF-VD *as is*, implying that our framework does not change the priority ordering of jobs determined by EDF-VD. Instead, our framework determines the portion of each job's execution to be scheduled when the priority of each job becomes the highest, i.e., it is possible for an LC job to execute in HC mode, and for an HC job to execute for more than its LC WCET in LC mode. On the other hand, EDF-VD without our framework restricts each job's execution to at most its LC WCET in LC mode and each HC and LC job's execution, respectively, to at most HC WCET and zero in HC mode.

Now, let's describe how our physical-state-aware dynamic slack management framework works under EDF-VD.

Runtime slack update. Our dynamic slack management determines when to update LC-mode and HC-mode slacks based on the following two observations.

O1. Upon release of a job $J_{i,j}$ of HC task $\tau_i \in \tau^H$, it is sufficient to assign $C_i^L(s_{i,j})$ and $C_i^H(s_{i,j})$ of resource in LC and HC modes (not $C_i^{L,\max}$ and $C_i^{H,\max}$), respectively, in order to meet $J_{i,j}$'s deadline. At the release time of a job $J_{p,q}$ of LC task τ_p, it is sufficient to assign at least $C_p^L(s_{p,q})$ of resource in LC-mode in order to meet $J_{p,q}$'s deadline in LC mode (not $C_p^{L,\max}$).

O2. Upon completion of $J_{i,j}$'s execution of any task, we can compare the actual execution time with its worst-case time allotment and reclaim any unused resources that were alloted to $J_{i,j}$.

Since we cannot know the physical state of a job until it is released, we need to assume that the resource demand for each future job of task τ_i is up to $C_i^{L,\max}$ and $C_i^{H,\max}$ in LC and HC modes, respectively to guarantee no deadline miss. However, with O1, upon release of a job, we can allocate a smaller amount of resource according to its physical state and reclaim the remaining portion of pre-allocated resources. Then, due to the sustainability property [25], it can still guarantee the MC-Schedulability with a reduced resource allocation for the released job. In addition, with O2, upon completion of a job, we can further reclaim the unused portion of assigned resources.

Based on the above two observations, LC-mode and HC-mode slacks are updated (i) when a new job is released (`JOB RELEASE`), or (ii) when a job is completed (`JOB COMPLETION`) as shown in Algorithm 1 (Lines 2–11). We keep track of the worst-case remaining execution time, RC_i, for the active job $J_{i,j}$ of τ_i. This is set to $C_i^M(s_{i,j})$ on `JOB RELEASE` (Line 6), where $M \in \{LC, HC\}$, decremented as the job executes (Lines 15, 27, and 30), and set to 0 on `JOB COMPLETION` (Line 10). Upon `JOB RELEASE` event of $J_{i,j}$, the amount of resource reclaimed in $[r_{i,j}, d_{i,j})$ during LC mode is $C_i^{L,\max} - C_i^L(s_{i,j})$ ($C_i^{H,\max} - C_i^H(s_{i,j})$ during HC mode). Upon `JOB COMPLETION` event of $J_{i,j}$, the amount of resource reclaimed in $[r_{i,j}, d_{i,j})$ during LC mode is $C_i^L(s_{i,j}) - AC_{i,j}$ ($C_i^H(s_{i,j}) - AC_{i,j}$ during HC mode), where $AC_{i,j}$ is the actual execution time of $J_{i,j}$. Upon each event, we update LC-mode or HC-mode slack for the interval of $[t_{cur}, d_1(t_{cur}))$, where t_{cur} is the current time instant and $d_1(t_{cur})$ is the earliest absolute deadline among all tasks' jobs whose deadline is after t_{cur} (Lines 8 and 11).[2] This way, it is possible to efficiently manage resource by updating available slacks according to the physical state. We will describe how to derive a safe lower-bound on the slacks in the following subsection.

Runtime slack scheduling. Now, let's describe how to utilize slacks in each mode. Under the classic MC scheduling model [1], a mode-switch from LC to HC occurs when a single HC job executes beyond its LC-part execution, and then all LC jobs will be immediately dropped upon a mode-switch even though there is a way to postpone a mode-switch and/or reduce LC job drops. With our dynamic slack management, the mode-switch can be postponed by assigning LC-mode slack to the HC task that executes more than its LC-part execution. In addition, the number of dropped LC jobs can be significantly reduced by assigning HC-mode slack to LC jobs after the mode-switch.

We present our slack-based mode-switch mechanism in Algorithm 1 (Lines 12–37); the basic idea is to utilize LC- and HC-mode slacks, inspired by Lemmas 2 and 3.

- In LC mode:
  1) If an HC job $J_{i,j}$ executes for $C_i^L(s_{i,j})$ time units in total but does not complete, then use $S_{LC}(t_{cur}, d_{i,j})$ to execute $J_{i,j}$ until $S_{LC}(t_{cur}, d_{i,j})$ becomes zero (Line 18).
  2) If $S_{LC}(t_{cur}, d_{i,j}) = 0$ but $J_{i,j}$ does not complete after its LC-part execution, then trigger a mode-switch to HC mode (Lines 20 and 21).
- In HC mode:
  1) If an LC job $J_{i,j}$ is the highest priority job at the current time instant t_{cur}, then use $S_{HC}(t_{cur}, d_{i,j})$ to execute $J_{i,j}$ until $S_{HC}(t_{cur}, d_{i,j})$ becomes zero (Line 30).
  2) If $S_{HC}(t_{cur}, d_{i,j}) = 0$ but $J_{i,j}$ does not complete, then drop the job (Line 32).
  3) After an idle instant, reset the mode to LC (Line 35).

In Section V-D, we will prove an significant property of Algorithm 1: its guarantee on no job deadline miss.

C. Slack Calculation

Now, let's consider how to calculate LC-mode and HC-mode slacks. Our goal is to find the maximum amount of slack time, which may be available during the interval $[t_{cur}, d_1(t_{cur}))$ in each mode, while guaranteeing all future deadlines ($\geq t_{cur}$) to be met based on Definition 1. By doing so, in LC mode, a currently executing HC job (having the earliest deadline) can use as much LC-mode slack as possible when it executes beyond its LC-part execution so that a mode-switch can be delayed to the maximum possible extent. Likewise, in HC mode, HC-mode slack can be used for LC jobs so that LC job drops can be postponed as late as possible and be reduced.

Algorithm 2 presents our slack calculation method, which is called by Lines 8, 11 and 21 of Algorithm 1. At time t_{cur}, we look at the interval until the earliest absolute deadline $d_1(t_{cur})$ among all tasks, try to defer as much execution as possible beyond $d_1(t_{cur})$, and compute the minimum amount of execution p that must execute before $d_1(t_{cur})$ in order to meet all future deadlines. Then, the slack is set to the remaining time slots except for p over the interval $[t_{cur}, d_1(t_{cur}))$. To calculate p, we use the similar approach as proposed for RT-DVS [26]. The underlying principle behind our slack calculation method is that the EDF algorithm will determine a feasible schedule if the utilization is less than or equal to 1.0 at any time [27].

[2]Note that if the corresponding job is an HC job in LC mode, the virtual deadline is applied to its absolute deadline; otherwise, the plain deadline is applied.

Algorithm 1 Dynamic Slack Management Framework

1: **Input:** t_{cur}, $M \in \{LC, HC\}$
2: Upon JOB RELEASE $(J_{i,j})$:
3: **if** $M = HC$ and $J_{i,j}$ is an LC job **then**
4: set $RC_i = 0$
5: **else**
6: set $RC_i = C_i^M(s_{i,j})$
7: **end if**
8: Update-slack($S_M(t_{cur}, d_1(t_{cur}))$)
9: Upon JOB COMPLETION $(J_{i,j})$:
10: set $RC_i = 0$
11: Update-slack($S_M(t_{cur}, d_1(t_{cur}))$)
12: During JOB EXECUTION $(J_{i,j})$:
13: **if** $M = LC$ **then**
14: **if** $RC_i > 0$ **then**
15: decrement RC_i
16: **else**
17: **if** $S_{LC}(t_{cur}, d_1(t_{cur})) > 0$ **then**
18: decrement $S_{LC}(t_{cur}, d_1(t_{cur}))$
19: **else**
20: Mode-switch(HC)
21: Update-slack($S_M(t_{cur}, d_1(t_{cur}))$)
22: **end if**
23: **end if**
24: **end if**
25: **if** $M = HC$ **then**
26: **if** $J_{i,j}$ is an HC job **then**
27: decrement RC_i
28: **else if** $J_{i,j}$ is an LC job **then**
29: **if** $S_{HC}(t_{cur}, d_1(t_{cur})) > 0$ **then**
30: decrement $S_{HC}(t_{cur}, d_1(t_{cur}))$ and RC_i
31: **else**
32: drop $J_{i,j}$
33: **end if**
34: **else if** Idle **then**
35: Mode-switch(LC)
36: **end if**
37: **end if**

Algorithm 2 Update-slack function

1: **Updating LC-mode slack:**
2: $U = U_{\tau^L}^L + \frac{1}{x} \cdot U_{\tau^H}^L$
3: $p = 0$
4: **for** $i = n$ to 1, $\tau_i \in \{\tau_1, ..., \tau_n | d_1(t_{cur}) \leq \cdots \leq d_n(t_{cur})\}$ **do**
5: {In reverse EDF order of tasks}
6: **if** $\tau_i \in \tau^L$ **then**
7: $U = U - u_i^{L,\max}$
8: **end if**
9: **if** $\tau_i \in \tau^H$ **then**
10: $U = U - \frac{u_i^{L,\max}}{x}$
11: **end if**
12: $q_i = \max\left(0, RC_i - (1 - U) \cdot (d_i(t_{cur}) - d_1(t_{cur}))\right)$
13: $U = \min\left(1.0, U + \frac{RC_i - q_i}{d_i(t_{cur}) - d_1(t_{cur})}\right)$
14: $p = p + q_i$
15: **end for**
16: $S_{LC}^*(t_{cur}, d_1(t_{cur})) = d_1(t_{cur}) - t_{cur} - p$
17: **Updating HC-mode slack:**
18: $U = x \cdot U_{\tau^L}^L + U_{\tau^H}^H$
19: $p = 0$
20: **for** $i = n$ to 1, $\tau_i \in \{\tau_1, ..., \tau_n | d_1(t_{cur}) \leq \cdots \leq d_n(t_{cur})\}$ **do**
21: {In reverse EDF order of tasks}
22: **if** $\tau_i \in \tau^H$ **then**
23: $U = U - u_i^{H,\max}$
24: $q_i = \max\left(0, RC_i - (1 - U) \cdot (d_i(t_{cur}) - d_1(t_{cur}))\right)$
25: $U = \min\left(1.0, U + \frac{RC_i - q_i}{d_i(t_{cur}) - d_1(t_{cur})}\right)$
26: $p = p + q_i$
27: **end if**
28: **end for**
29: $S_{HC}^*(t_{cur}, d_1(t_{cur})) = d_1(t_{cur}) - t_{cur} - p$

For LC-mode slack $S_{LC}(t_{cur}, d_1(t_{cur}))$, we examine all tasks in reverse EDF order, i.e., latest deadline first (Line 4). Note that tasks are indexed in EDF order (i.e., for τ_i and τ_k where $i < k$, $d_i(t_{cur}) \leq d_k(t_{cur})$). We assume that future job invocations of all tasks require the worst-case utilization under EDF-VD, which is $U_{\tau^L}^L + \frac{U_{\tau^H}^L}{x}$ (Line 2). Under the assumption, we allocate τ_i's remaining execution RC_i between the earliest deadline $d_1(t_{cur})$ and its own deadline d_i until the total utilization in the interval no greater than 1, and compute the minimum amount of execution (denoted as q_i) that must be allocated before $d_1(t_{cur})$ in order to finish by its own deadline (Line 12). A cumulative utilization U is adjusted to reflect the actual utilization of τ_i after $d_1(t_{cur})$ (Line 13). This step is repeated for all tasks. p is simply the sum of the q_i values for all tasks, and therefore reflects the total amount of execution that must be done by $d_1(t_{cur})$ in order to meet their deadlines. Then, we can calculate a lower bound of $S_{LC}(t_{cur}, d_1(t_{cur}))$ (denoted by $S_{LC}^*(t_{cur}, d_1(t_{cur}))$), by subtracting p from the total interval length (Line 16).[3]

For HC-mode slack $S_{HC}(t_{cur}, d_1(t_{cur}))$, only HC tasks contribute to p under the assumption that future job invocations of HC tasks require the worst-case utilization of $U_{\tau^H}^H$. Then, a lower bound of HC-mode slack $S_{HC}(t_{cur}, d_1(t_{cur}))$ (denoted by $S_{HC}^*(t_{cur}, d_1(t_{cur}))$) can be calculated in accordance with the same procedure as shown above.

Note that LC- and HC-mode slacks are updated upon either JOB RELEASE or JOB COMPLETION to reflect any change of physical state or early completion. When a new job $J_{i,j}$ is released and becomes active, its remaining execution time is assigned according to its physical state $s_{i,j}$, and the slack values are updated to reflect this. Therefore, our dynamic slack management enables to capture how the change in the physical state of each task affects LC- and HC-mode slacks and efficiently utilize available slacks at each criticality mode to delay a mode-switch or LC job drops.

D. Analysis of Dynamic Slack Management

In this subsection, we will prove an important property of Algorithm 1, which guarantees MC-schedulability.

[3] An example of illustrating slack calculation presented in Algorithm 2 is provided in Appendix B (see Fig. 4) of the supplement file [24].

First, the following two lemmas state that Algorithm 1 does not cause any deadline miss in LC and HC modes, respectively.

Lemma 5: Suppose that τ is scheduled by Algorithm 1 with EDF-VD. Also, suppose that Eq. (4) holds, MC-schedulability is guaranteed until t_{cur}, and the system is in LC mode at t_{cur}. Let J_1 (if any) denote a job which performs its HC-part execution for at most $S^*_{LC}(t_{cur}, d_1(t_{cur})) > 0$ in $[t_{cur}, d_1(t_{cur}))$. If every job except J_1 performs its LC-part execution (and does not perform its HC-part execution) after t_{cur}, there is no job deadline miss.

Proof: We apply the fact that there is no job deadline miss under EDF if the total utilization at any time is not greater than 1.0 [27], to the first inequality in Eq. (4) (originally from [7]). Then, what we need to prove is that Algorithm 1 with EDF-VD satisfies $U^L_{\tau_L}(t) + \frac{1}{x} \cdot U^L_{\tau_H}(t) \leq 1.0$ for every t, where $U^L_{\tau_L}(t)$ and $U^L_{\tau_H}(t)$ denote the sum of run-time LC utilization of LC tasks and HC tasks, respectively, each of whose release time is before t and deadline is after t.

Starting with setting the total static utilization of $U^L_{\tau_L} + \frac{1}{x} \cdot U^L_{\tau_H}$ in Line 2 of Algorithm 2, the algorithm updates the run-time utilization by Line 7 or 10, and Lines 12-13, and calculates the largest q_i that does not compromise $U^L_{\tau_L}(t) + \frac{1}{x} \cdot U^L_{\tau_H}(t) \leq 1.0$ in Line 12, implying that the lemma holds.

Note that one may wonder why there is no scaling factor $\frac{1}{x}$ (indicating the virtual deadline) in Lines 12-13 for HC tasks. This is because, d_i is defined as the *virtual deadline* of HC jobs, so the scaling factor is already reflected in Lines 12-13. ■

Lemma 6: Suppose that τ is scheduled by Algorithm 1 with EDF-VD. Also, suppose that Eq. (4) holds, MC-schedulability is guaranteed until t_{cur}, and the system is in HC mode at t_{cur}. Let J_1 (if any) denote an LC job which performs its LC-part execution for at most $S^*_{HC}(t_{cur}, d_1(t_{cur})) > 0$ in $[t_{cur}, d_1(t_{cur}))$. If every HC job executes for at most its HC WCET and every LC job except J_1 does not execute at all after t_{cur}, there is no HC job deadline miss.

Proof: The proof is similar to that of Lemma 5. We now prove that Algorithm 1 with EDF-VD satisfies $x \cdot U^L_{\tau_L}(t) + U^H_{\tau_H}(t) \leq 1.0$ for every t, from [27] and the second inequality of Eq. (4) (originally from [7]), where $U^L_{\tau_L}(t)$ and $U^H_{\tau_H}(t)$ denote the sum of run-time LC utilizations of LC tasks and that of run-time HC utilizations of HC tasks, respectively, each of whose release times is before t and deadline is after t.

Starting with setting the total static utilization of $U^H_{\tau_H} + x \cdot U^L_{\tau_L}$ in Line 18 of Algorithm 2, the algorithm updates the run-time utilization by Lines 23-25, and calculates the largest q_i that does not compromise $x \cdot U^L_{\tau_L}(t) + U^H_{\tau_H}(t) \leq 1.0$ in Line 24, implying the lemma holds. ■

Combining the above two lemmas, the following theorem states the MC-schedulability of Algorithm 1 with EDF-VD.

Theorem 1: Suppose that Eq. (4) holds for τ. Then, Algorithm 1 with EDF-VD guarantees no job deadline miss.

Proof: By Lemmas 5 and 6, the theorem holds. ■

Runtime complexity for dynamic slack management. For EDF-VD alone, a scheduler is invoked upon arrival/preemption/completion of a job or a mode-switch. For our proposed framework, the additional operations are summarized as follows: 1) upon arrival/completion of a job or a mode-switch, our framework updates the slack by Algorithm 2 with the complexity of $O(n)$, where n is the number of tasks; 2) whenever an HC job requests more than its LC-part execution in LC mode (which is the same instant of a mode-switch event happened in EDF-VD), our framework allocates the slack to the job with the complexity of $O(1)$; and 3) whenever LC- and HC-mode slacks become zero, our framework triggers a mode-switch. Therefore, our framework requires $O(n)$ runtime complexity at job arrival/completion and mode-switch instants, and generate one more event than EDF-VD. Note that EDF-VD can be implemented with the runtime complexity of $O(logn)$ upon the same event that our framework works on (i.e., arrival/completion of a job or mode-switch). The additional runtime overhead of our framework beyond EDF-VD comes from the updating of slacks. We use an approach similar to [17] for slack calculation. According to [17], the computation cost of slack calculation is negligible, assuming the scheduler provides a priority queue. We can also bound the number of slack updates per job, thus accounting for the overhead of slack calculation by adding it to the WCET value of each task.

VI. Evaluation

We now demonstrate a significant improvement of resource-efficiency by the proposed physical-state-aware dynamic slack management framework. We focus on resource-efficiency in terms of the ratio of dropped LC jobs to all released LC jobs. Therefore, we will look at how long a mode-switch can be postponed and the number of LC jobs that can be executed in HC mode without dropping. Note that we do not compare the MC-schedulability since our approach employs the existing virtual deadline assignment and does not change the priority ordering of jobs determined by EDF-VD, so we can achieve the same MC-schedulability as EDF-VD can have.

We first show the experimental results for our case study of an ADAS system presented in Section II. Then, we show the extensive simulation results for synthetic task sets with randomly generated parameters.

We compare the following three different approaches:

- Base: EDF-VD with the classic MC task model [1],
- Base-PHY: EDF-VD with the physical-state-aware MC task model, and
- Base-PHY-DSM: EDF-VD with the physical-state-aware dynamic slack management framework.

Under Base, all jobs of each task τ_i are always assigned their resources based on $C_i^{L,\max}$ and $C_i^{H,\max}$ regardless of the physical state and are scheduled under EDF-VD. Under Base-PHY, all jobs are scheduled under EDF-VD, and a mode-switch of each HC job $J_{i,j}$ is determined based on $C_i^L(s_{i,j})$ without slack usage. Under Base-PHY-DSM, all jobs are assigned their resources according to their physical states and are scheduled according to the physical-state-aware dynamic slack management framework presented in Algorithm 1.

The following metrics are used to evaluate the performance of the listed approaches above:

- R_{drop}: the percentage of dropped LC jobs over all released LC jobs,
- N_{ms}: the number of mode-switch occurred,
- I_{LC}: the average length of LC-mode interval between mode-switch, and
- S_{used}: the amount of slack used for task execution.

A. Case Study: an ADAS system

We consider that both ACC and AVS are HC tasks and run with other 4 LC tasks together.[4] For simplicity, we assume that the physical state of ACC or AVS can be divided into two, i.e., either steady or transient states. However, our physical-state-aware MC task model and dynamic slack management framework can be applied to multiple physical states without loss of generality. The physical state of each component changes independently according to its sensing data and control output as shown in Fig. 1. The periods of ACC and AVS are set to $100ms$. The LC and HC WCET estimates at each physical state are set to the average- and worst-case number of iterations, respectively. We assume that each iteration requires up to $2ms$. The actual execution time traces for ACC and AVS are obtained from a real driving scenario where a host vehicle follows a reference trajectory including occasional double lane change maneuvers when the preceding vehicle travels at a varying speed. In addition, we generate 4 LC tasks according to the same procedure presented in Section VI-B, and their parameters are shown in Table I.

We run a simulation for 80 seconds and compare R_{drop}, N_{ms}, and I_{LC} for Base and Base-PHY-DSM as shown in Table II.[5] Under Base, 13.7% of LC jobs are dropped, while Base-PHY-DSM drops only 0.6% of LC jobs in HC mode. Under Base-PHY-DSM, the number of mode-switches occurred (N_{ms}) is reduced by 16x, and the system operates in LC mode 17x longer on average between mode-switch (I_{LC}), by utilizing 1,753 time units of slack as compared to Base. Consequently, Base-PHY-DSM drops 21x less LC jobs than Base does while guaranteeing MC-schedulability. Such an improvement can be interpreted as the benefit of capturing dynamic execution behavior according to the physical state (by our physical-state-aware MC task model) when reclaiming available resources and utilizing them corresponding to the criticality mode on the fly (by our dynamic slack management). As a result, Base-PHY-DSM can delay a mode-switch as much as possible and significantly reduce the number of dropped LC jobs.

[4]The task set configuration of our case-study is practical and representative of ADAS. In an ADAS system, longitudinal and lateral controllers, i.e., ACC and AVS, are key components, which are usually implemented with other supporting components, such as monitoring and display tasks, which might be considered as LC tasks [28].

[5]Note that Base-PHY is excluded for the case study since the result is same as Base. This is because the iteration count is alway one in the steady state for both ACC and AVS cases according to our experiment, so a mode-switch does not occur in the steady state under either Base or Base-PHY. Considering the transient state, Base and Base-PHY allocate the same amount of resources to ACC and AVS tasks. However, the simulation results of Base-PHY are included for synthetic task sets.

TABLE I. LC TASK PARAMETERS

(ms)	LC task 1	LC task 2	LC task 3	LC task 4
Period T_i	200	200	80	50
$C_i^L(s_i)$	$\{61, 17\}$	$\{35, 10\}$	$\{5, 2\}$	$\{7, 3\}$

TABLE II. SIMULATION RESULTS OF A CASE STUDY

	R_{drop}	N_{ms}	I_{LC}	S_{used}
Base	13.7	201	388	0
Base-PHY-DSM	0.6	12	6637	1753

B. Extensive simulations

Task set generation. We generate a synthetic task set similarly in [4, 5], which can be summarized as follows. The number of tasks is chosen among 4, 6, and 8. Each task is generated based on the following parameters. Task τ_i is an HC task with probability of 0.5 and has two physical states, i.e., $s_i \in \{a_i, b_i\}$. Period T_i is chosen in $\{20, 25, 40, 50, 80, 100, 200, 250, 400\}$ as a representative setting for automotive and avionics systems [4, 5]. $C_i = \{C_i^L(a_i), C_i^H(a_i), C_i^L(b_i), C_i^H(b_i)\}$ is determined by using the UUniFast algorithm [29]. In particular, $u_i^L(a_i)$ of each task τ_i is randomly generated such that $\sum_i u_i^L(a_i) = 0.7$. Then, $u_i^L(b_i)$ is uniformly chosen in $[1, PF \cdot u_i^L(a_i)]$, where $PF = 2$, and $u_i^H(s_i)$ is set to $CF \cdot u_i^L(s_i)$, where $CF = 2$.

We generate 100 task sets for each configuration of the number of tasks (4, 6, and 8). Note that we only generate MC-schedulable task sets under EDF-VD satisfying Eq. (4). For each task set, we run a simulation for 100,000 time units with the following runtime configuration. We set the probability of the physical state change for a task ($P(Phy)$) to 0.1. At a job release, we determine the physical state $s_{i,j}$ of a job $J_{i,j}$ based on $P(Phy)$. We also set the probability of showing HC behavior for an HC task ($P(HC)$) to 0.1. If a job $J_{i,j}$ shows HC behavior, then the actual execution time of the job is uniformly chosen in $[C_i^L(s_{i,j}), C_i^H(s_{i,j})]$, else it is uniformly chosen in $[0.7 \cdot C_i^L(s_{i,j}), C_i^L(s_{i,j})]$.

Simulation results. Table III shows the simulation results. When the number of tasks (n) is 4, Base, Base-PHY, and Base-PHY-DSM drop 5.4%, 8.6%, and 0.6% LC jobs, respectively. Note that Base-PHY drops more LC jobs than Base does. Under Base-PHY, a mode-switch of each HC job $J_{i,j}$ is triggered based on $C_i^L(s_{i,j})$, that is less than or equal to $C_i^{L,\max}$, so a mode-switch is more likely to occur than Base, where a mode-switch is determined based on $C_i^{L,\max}$. As we can see in Table III, the number of mode-switches under Base-

TABLE III. SIMULATION RESULTS OF SYNTHETIC TASK SETS

n		R_{drop}	N_{ms}	I_{LC}	S_{used}	N_{job}
4	Base	5.4	157	2105	0	7878
	Base-PHY	8.6	272	971	0	
	Base-PHY-DSM	0.6	25	15146	894	
6	Base	5.3	190	1069	0	11117
	Base-PHY	8.9	327	521	0	
	Base-PHY-DSM	0.2	14	21139	944	
8	Base	7.1	276	500	0	14743
	Base-PHY	11.3	459	302	0	
	Base-PHY-DSM	0.2	10	28561	1045	

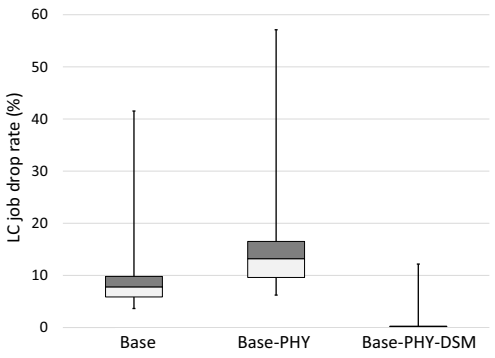


Fig. 2. The percentage of dropped LC jobs over all released LC jobs

PHY is increased by 1.7x compared to Base. This implies that, without a proper slack reclamation method, using the physical-state-aware MC task model can result in even worse performance. However, with our dynamic slack management framework, Base-PHY-DSM effectively utilizes 894 time units of LC-mode and HC-mode slacks (0.9% of the total simulation time) to reduce the number of mode-switches and the number of dropped LC jobs. Therefore, Base-PHY-DSM reduces the LC job drop rate by 9x and 14x of Base and Base-PHY, respectively. A similar trend can be seen for $n = 6$ and 8.

One interesting observation is that the performance of Base and Base-PHY is getting worse as the number of tasks increases, while Base-PHY-DSM shows the opposite trend. For example, as n increases from 4 to 8, R_{drop} of Base is increased from 5.4% to 7.1%, but R_{drop} of Base-PHY-DSM is decreased from 0.6% to 0.2%. As n increases, the number of total jobs (N_{job}) increases as shown in the last column of Table III. Then, there exist more HC jobs that can potentially show HC behavior. Therefore, the number of mode-switches (N_{ms}) under Base and Base-PHY increases from 157 to 276 and from 272 to 459, respectively, as n increases, leading to more LC job drops. In contrast, under Base-PHY-DSM, it is more effective to utilize slack as n increases. Since we generate task sets to have the same utilization of $\sum_i u_i^L(a_i) = 0.7$ for all configuration of the number of tasks, each task in a task set has less utilization as n increases. Therefore, we can execute more jobs with the same amount of slack, leading to less LC job drops under Base-PHY-DSM.

The percentage of dropped LC jobs for all cases of $n = 4$, 6, and 8 are shown in Fig. 2. The box in Fig. 2 represents the range of values between quartiles (25 and 75 percentiles). The horizontal line in the middle of the box is the median. The vertical line shows the 5 and 95 percentiles. The average values for Base, Base-PHY, and Base-PHY-DSM are 5.9%, 9.6%, and 0.3%, respectively, exhibiting 20x difference between Base and Base-PHY-DSM.

VII. Related Work

Since Vestal's pioneering work [1] on MC systems, a large number of studies have been done on MC real-time scheduling. Many existing solutions [7, 30–32] share the pessimistic strategy in which all LC jobs will be dropped immediately once the system switches to HC mode. Other studies provide a degraded service to LC tasks after a mode-switch, stretching their periods [33, 34], lowering their priorities [33], or allowing the execution of some LC jobs [35–37]. All of these studies, however, use static resource allocation in that once a single HC task executes beyond its LC-part execution, the system switches to HC mode regardless of the amount of resources available.

Some of recent studies [2–6] support dynamic resource allocation, exploiting slack to reduce the pessimism of the previous static approaches. Nevertheless, most studies [2–4] consider a statically available slack derived by offline schedulability tests. Santy *et al.* [2] proposed a method to calculate a slack offline, allowing LC tasks to proceed with their execution without compromising HC task execution. The bailout protocol [4] utilizes the offline slack for a timely return to LC mode in order to reduce the negative impact on LC tasks. Niz and Phan [3] proposed a slack-based scheduling algorithm for multi-modal mixed-criticality systems. Although they considered a multi-mode environment in which some tasks change their parameters, they assumed system-wide sequential mode-changes. In contrast, we consider the situation where each task may show a different execution behavior according to each physical state, and hence multiple changes of each task's execution mode may take place over consecutive jobs.

There have been a few studies [5, 6] that focus on dynamic slack at runtime. Hu *et al.* [5] considered a runtime available slack to delay a mode-switch. Gu and Easwaran [6] proposed dynamic budget management that determines the amount of low-criticality execution budget for each HC task at runtime by considering the dynamic slack. They assumed that the total amount of low-criticality execution budgets for all HC tasks is statically assigned. Although those solutions utilize the dynamic slack to delay a mode-switch as much as possible, none of them considered varying physical states when calculating the dynamic slack.

VIII. Conclusion

In this paper, we formulated the problem of efficient utilization of cyber resources under dynamically-changing physical states in MC systems. To solve the problem, we introduced a new MC task model that captures different physical states, and then proposed slack concepts for the new MC task model. Utilizing the model and the slack concepts, we developed a physical-state-aware dynamic slack management framework for EDF-VD, and demonstrated its efficiency in achieving a significant reduction of the number of dropped LC jobs over EDF-VD, via a case study and in-depth evaluation. In future, we would like to consider physical-state-aware dynamic virtual deadline assignment in order to explore the possibility of enhancing MC-schedulability as well as resource-efficiency.

Acknowledgment

We are grateful to Shige Wang of GM R&D Center for helpful comments. The work reported in this paper was supported in part by the Office of Naval Research under Grants N00014-15-1-2163 and N00014-18-1-2141, and the National Science Foundation under Grant CNS-1329702. This

work was also supported in part by the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (2017R1A2B2002458, 2017H1D8A2031628, 2017K2A9A1A01092689).

REFERENCES

[1] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *RTSS*, 2007.

[2] F. Santy, L. George, P. Thierry, and J. Goossens, "Relaxing mixed-criticality scheduling strictness for task sets scheduled with FP," in *ECRTS*, 2012.

[3] D. de Niz and L. T. Phan, "Partitioned scheduling of multi-modal mixed-criticality real-time systems on multiprocessor platforms," in *RTAS*, 2014.

[4] I. Bate, A. Burns, and R. I. Davis, "A bailout protocol for mixed criticality systems," in *ECRTS*, 2015.

[5] B. Hu, K. Huang, P. Huang, L. Thiele, and A. Knoll, "On-the-fly fast overrun budgeting for mixed-criticality systems," in *EMSOFT*, 2016.

[6] X. Gu and A. Easwaran, "Dynamic budget management with service guarantees for mixed-criticality systems," in *RTSS*, 2016.

[7] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie, "The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems," in *ECRTS*, 2012.

[8] D. Hrovat, S. D. Cairano, H. Tseng, and I. Kolmanovsky, "The development of model predictive control in automotive industry: A survey," in *CCA*, 2012.

[9] Adaptive cruise control system using model predictive control. [Online]. Available: https://www.mathworks.com/help/mpc/examples/design-an-adaptive-cruise-control-system-using-model-predictive-control.html

[10] Autonomous vehicle steering using model predictive control. [Online]. Available: https://www.mathworks.com/help/mpc/examples/autonomous-vehicle-steering-using-model-predictive-control.html

[11] M. S. K. Lau, S. P. Yue, K. V. Ling, and J. M. Maciejowski, "A comparison of interior point and active set methods for FPGA implementation of model predictive control," in *ECC*, 2009.

[12] J. Kim, K. Lakshmanan, and R. Rajkumar, "Rhythmic tasks: A new task model with continually varying periods for cyber-physical systems," in *ICCPS*, 2012.

[13] G. C. Buttazzo, E. Bini, and D. Buttle, "Rate-adaptive tasks: Model, analysis, and design issues," in *DATE*, 2014.

[14] A. Biondi, A. Melani, M. Marinoni, M. D. Natale, and G. Buttazzo, "Exact interference of adaptive variable-rate tasks under fixed-priority scheduling," in *ECRTS*, 2014.

[15] R. I. Davis, T. Feld, V. Pollex, and F. Slomka, "Schedulability tests for tasks with variable rate-dependent behaviour under fixed priority scheduling," in *RTAS*, 2014.

[16] Z. Guo and S. K. Baruah, "Uniprocessor edf scheduling of avr task systems," in *ICCPS*, 2015.

[17] D. de Niz, L. Wrage, N. Storer, A. Rowe, and R. Rajkumar, "On resource overbooking in an unmanned aerial vehicle," in *ICCPS*, 2012.

[18] D. Liberzon and A. S. Morse, "Basic problems in stability and design of switched systems," *IEEE Control Systems Magazine*, vol. 19(5), pp. 59–70, 1999.

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

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

[21] J. P. Lehoczky and S. Ramos-Thuel, "An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems," in *RTSS*, 1992.

[22] R.I.Davis, K.W.Tindell, and A. Burns, "Scheduling slack time in fixed priority preemptive systems," in *RTSS*, 1993.

[23] R. Jejurikar and R. Gupta, "Dynamic slack reclamation with procrastination scheduling in real-time embedded systems," in *DAC*, 2005.

[24] H. S. Chwa, K. G. Shin, H. Baek, and J. Lee, "Supplement of "physical-state-aware dynamic slack management for mixed-criticality systems"," https://kabru.eecs.umich.edu/wordpress/wp-content/uploads/CSB18sub.pdf.

[25] S. Baruah and A. Burns, "Sustainable scheduling analysis," in *RTSS*, 2006.

[26] P. Pillai and K. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *Proceedings of ACM Symposium on Operating Systems Principles*, 2001, pp. 89–102.

[27] S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson, "Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes," in *RTSS*, 2003.

[28] L. Vlacic, M. Parent, and F. Harashima, *Intelligent Vehicle Technologies: Theory and Applications*. Butterworth-Heinemann, 2001.

[29] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30(1-2), pp. 129–154, 2005.

[30] P. Ekberg and W. Yi, "Bounding and shaping the demand of mixed-criticality sporadic tasks," in *ECRTS*, 2012.

[31] S. Baruah, A. Burns, and R. Davis, "Response-time analysis for mixed criticality systems," in *RTSS*, 2011.

[32] A. Easwaran, "Demand-based scheduling of mixed-criticality sporadic tasks on one processor," in *RTSS*, 2013.

[33] A. Burns and S. Baruah, "Towards a more practical model for mixed criticality systems," in *WMC, RTSS*, 2013.

[34] H. Su and D. Zhu, "An elastic mixed-criticality task model and its scheduling algorithm," in *DATE*, 2013.

[35] J. Lee, H. S. Chwa, L. T. Phan, I. Shin, and I. Lee, "MC-ADAPT: Adaptive task dropping with task-level mode switch in mixed-criticality scheduling," in *EMSOFT*, 2017.

[36] X. Gu, A. Easwaran, K.-M. Phan, and I. Shin, "Resource efficient isolation mechanisms in mixed-criticality scheduling," in *ECRTS*, 2015.

[37] J. Ren and L. T. X. Phan, "Mixed-criticality scheduling on multiprocessors using task grouping," in *ECRTS*, 2015.