ADAPTIVE: Agent-Based Learning for Bounding Time in Mixed-Criticality Systems

Behnaz Ranjbar, Ali Hosseinghorban, and Akash Kumar, *Senior Member, IEEE*
Chair of Processor Design, CFAED, Technische Universität Dresden, Dresden, Germany
{behnaz.ranjbar, akash.kumar}@tu-dresden.de, ali.hosseinghorban1394@sharif.edu

***Abstract*—In Mixed-Criticality (MC) systems, the high Worst-Case Execution Time (WCET) of a task is a pessimistic bound, the maximum execution time of the task under all circumstances, while the low WCET should be close to the actual execution time of most instances of the task to improve utilization and Quality-of-Service (QoS). Most MC systems consider a static low WCET for each task which cannot adapt to dynamism at run-time. In this regard, we consider the run-time behavior of tasks and propose a learning-based approach that dynamically monitors the tasks' execution times and adapts the low WCETs to determine the ideal trade-off between mode-switches, utilization, and QoS. Based on our observations on running embedded real-time benchmarks on a real platform, the proposed scheme improves the QoS by 16.4% on average while reducing the utilization waste by 17.7%, on average, compared to state-of-the-art works.**

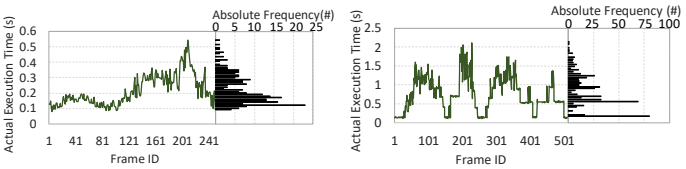
***Index Terms*—Mixed-Criticality, Mode Switching Probability, Machine Learning, Service Adaptation, WCET Analysis.**

I. Introduction

Mixed-Criticality (MC) systems integrate a large number of real-time tasks with different criticality levels onto a common hardware platform to meet stringent requirements such as cost, space, and timing [1]–[4]. Medical devices, automotive, and avionics are the most common safety-critical applications, evolving into MC systems [2], where the successful execution of tasks with Higher-Criticality levels (HC tasks) must be guaranteed in all circumstances to prevent catastrophic damages, while a higher number of Low-Criticality (LC) tasks should be executed to improve service requirements (i.e., Quality-of-Service (QoS)) and consequently, the processor utilization [3].

From the MC tasks' execution times perspective, multiple WCETs are determined corresponding to the multiple criticality levels [1]–[5]. A well-known type of MC system is a dual-criticality system (consisting of LC and HC tasks) in which two WCETs (low (C^{LO}) and high (C^{HI})) are determined. The C^{HI} of a task is a pessimistic bound, the maximum execution time of the task under all circumstances. However, this bound is high, and considering it to schedule the tasks leads to poor processor utilization and QoS (i.e., fewer LC tasks can be scheduled) [3]. To this end, MC systems consider a C^{LO} for HC tasks that should be close to the actual execution time of most task instances to improve utilization and QoS. At run-time, the system starts its operation in low-criticality mode (LO mode), and if the execution time of at least one HC task exceeds its C^{LO}, the system switches to the high-criticality mode (HI mode). To guarantee the correct execution of HC tasks in HI mode, C^{HI} are considered to schedule HC task. Since HC tasks may execute longer in HI mode compared to LO mode, the LC tasks are dropped/degraded to their minimum service requirements to guarantee the correct execution of HC tasks before their deadlines [2], [5], [6].

As can be realized, the low WCETs (C^{LO}) play an important role in improving the MC system's QoS. Determining the high values for C^{LO}s can minimize the mode switches but reduce the processor utilization due to scheduling fewer tasks. On the other hand, the utilization can be maximized by determining the low values for C^{LO}s, but with a high number of mode switches, which is not desirable. Although there are many approaches like what is presented in [7] and tools like OTAWA [8] to determine the C^{HI}, there are few approaches for determining the C^{LO}s in MC systems. These few approaches [1], [3], [5], [6] analyze the tasks at design-time, and set the constant WCETs for tasks in LO mode, which remain unchanged during run-time. Such static techniques can cause significant performance loss for LC tasks or processor under-utilization if the C^{LO}s are not close to actual execution times. In general, the actual execution time of tasks depends on their input values. Due to the spatial or temporal correlation in the input data stream like video, the execution times of the tasks are often temporally correlated.



(a) Input video with few objects to detect (b) Input video with few and many objects to detect

Fig. 1: Execution time values for two different time recording videos as input for Object Detection function during run-time and their time distribution. This figure shows that both aspects of run-time and design-time behavior should be considered in MC system design and task properties determination.

Motivational Example: Fig. 1 shows the computational times of the object detection function running on the ODROID XU4 board powered by ARM Cortex A7. Note that the object detection function is one of the main functions in an autonomous driving application – an MC system. For input, videos from a road camera in the two different time slots, converted to motion jpegs, are given to the function of detecting cars on the road. The videos were recorded for a period of time when it experienced both light and heavy traffic. Fig. 1 shows how the computation times of detecting objects vary during run-time. The computation time values in this function depend on the number of objects to be detected. As we can see, the times of multiple jpeg images are clustered due to the temporal correlation between the subsequent inputs presented to the application. For this example, static approaches such as the one presented in [1], [3], [5] set the static C^{LO}, considering the execution time of the majority of instances. This static WCET works fine for some time, but it may lead to frequent mode switches when there are many objects to detect (e.g., heavy traffic) or poor utilization when there are few objects to detect in this function (e.g., light traffic). As a result, proposing an adaptive scheme to determine C^{LO} dynamically may significantly improve the mode switches, QoS, and utilization. Therefore, the system's run-time behavior can be investigated

This work is supported by a grant from Software Campus through the German Federal Ministry of Education and Research, under the project SARA: Safety-Aware Relocation of functions in a multi-core computer Architecture.

TABLE I: A brief overview on the state-of-the-art MC approaches.

#	Related Works	Dynamic QoS-Aware	C^{LO} Adjustment	Design/Run Time	Use of ML
1	Baruah'12 [1], Liu'18 [5]	×	×	✓/×	×
2	Ranjbar'21 [3]	×	✓	✓/×	×
3	Gu'16 [4], Gu'18 [11] Hu'19 [12]	×	✓	×/✓	×
4	Su'16 [6]	✓	×	×/✓	×
5	Ranjbar'22 [13]	✓	×	×/✓	✓
6	ADAPTIVE	✓	✓	✓/✓	✓

by monitoring the execution times and adjusting C^{LO}.

In this work[1], we propose a novel learning-based run-time scheme for determining C^{LO} to 1) effectively reduce the system mode switches, 2) have high processor utilization and, consequently, a high value of QoS, 3) guarantee the system to be schedulable in each criticality level, 4) not be affected and varied by sudden changes of execution times. To the best of our knowledge, there is no method yet to determine C^{LO} of MC tasks at run-time based on the behavioral system changes while making a trade-off between the QoS and mode switches.

Contributions: The main contributions of this paper are:

- Presenting a novel adaptive scheme to analyze and obtain the low WCETs (C^{LO}) of MC tasks at run-time, and manage the mode switching probability and QoS.
- Proposing a learning-based mechanism, called *ADAPTIVE*, to design an adaptive MC system, and improve its timing behaviour at run-time.
- Presenting a dynamic QoS-aware scheduling algorithm to improve the results' quality at run-time based on the system changes, while guaranteeing the minimum service of LC tasks, even in the HI mode.

The rest of this paper is organized as follows. In Sections II and III, we provide an overview of related works, and the MC task and system operational models, respectively. The proposed scheme is explained in Section IV. Then, we analyze the experiments and conclude in Sections V, and VI, respectively.

II. Related Works

In the last decade, a significant number of papers have been published in MC system design and task scheduling. Burns and Davis [2] provided a comprehensive study in this field; however, in this section, we mostly focus on the works with the objectives of dynamic QoS improvement and WCET analysis at both design- and run-time, which are summarized in Table I.

Many recent papers have designed the MC systems by setting the C^{LO} at design-time which is not changing during the run-time (row 1 of Table I). As an example, in [1], [5], [10], the C^{LO}s are set as a percentage of C^{HI}s. These estimations are not accurate since WCET and actual execution times do not always have a linear relationship. In addition to this, researchers in [3] have recently proposed an approach to determine the C^{LO}s theoretically by using Chebyshev's theorem (row 2). In this paper, the author assumed that the inputs are random. However, in many applications which interact with the environment, the input remains the same for a while. So, since their approach is static, it cannot adapt to actual execution times at run-time. Besides, a few studies such as [4], [11], [12] (row 3) have focused on determining the C^{LO}s at run-time, based on their overall processing requirements and actual execution times. However, the goal of these methods is to postpone the mode switches for a long time while only guaranteeing a minimum QoS for LC tasks.

[1]An extended abstract of this article has been published in Design, Automation & Test in Europe Conference & Exhibition (DATE), 2023, with the title 'Motivating Agent-Based Learning for Bounding Time in Mixed-Criticality Systems' [9].

Some works such as [6], [13] (row 4) have considered the C^{LO} as a percentage of C^{HI} and improved the QoS at run-time by exploiting the accumulated dynamic slack generated by early completion of HC tasks. Since the dynamic slack is considered as a wrapper task with a deadline [6] and cannot be used anytime, these approaches do not use system utilization optimally to improve the QoS.

Therefore, a run-time system investigation and WCET analysis of MC tasks is needed to improve the confidence in WCET values, service adaptation, and utilization [2]. In this work, we propose an adaptive scheme based on machine-learning techniques to not just analyze the WCETs in LO mode but also monitor and control the system's behavior at run-time to make an ideal trade-off between processor utilization, QoS, and mode switches with the assumptions made during static analysis.

III. Mixed-Criticality Task Model

We consider real-time MC applications consisting of periodic independent tasks, executed on a preemptive uniprocessor. Analogous to [1], [3], [5], [10], a dual-criticality system is considered, in which a number of independent MC tasks $\{\tau_1, \tau_2, ..., \tau_n\}$ are executed. Each task τ_i is characterized as $\{\zeta_i, C_i^{LO}, C_i^{HI}, D_i, T_i\}$, where:

- ζ_i denotes the criticality level of τ_i ($\zeta_i \in \{LC, HC\}$)
- $C_i^{LO(HI)}$ denotes the WCET of τ_i in the LO (HI) mode
- D_i and T_i denote the deadline and period of τ_i, respectively, where $D_i = T_i$ in this article, analogous to [3], [5], [10]

In these MC systems, analogous to most of MC works, for each HC task, $C_i^{LO} \leq C_i^{HI}$. Since we use the task utilization values to check the MC task schedulability on processor, the utilization of task τ_i at criticality mode LO (HI) is defined as $u_i^{LO} = \frac{C_i^{LO}}{T_i}$ ($u_i^{HI} = \frac{C_i^{HI}}{T_i}$).

Initially, the system starts its operation in LO mode, where all HC tasks and LC tasks must be executed successfully before their deadlines. In this mode, if the execution time of at least one HC task exceeds its C^{LO}, the system switches to the HI mode. In this HI mode, the LC tasks are dropped/degraded to their minimum service requirements to guarantee the correct execution of HC tasks before their deadlines. If there is no ready HC task in the processor's queue, the system switches back safely to LO mode [1], [3], [5]. From the perspective of LC tasks' service adaptation, the QoS can be adjusted by controlling the rate of LC tasks' execution, i.e., the tasks' periods. In general, the system should release and execute the LC tasks by considering their actual period to improve the QoS and functionality with high precision of outputs [6]. The QoS is defined as $\frac{n_{LC}^{schd}}{n_{LC}^{max}}$ [13], [14], where n_{LC}^{schd} and n_{LC}^{max} are the number of LC tasks, that can be scheduled, and the number of all LC tasks in the system, respectively. Note that n_{LC}^{max} corresponds to the state that all LC tasks can regularly release with their actual period. Therefore, the minimum QoS can be employed by releasing the LC tasks with a larger period.

IV. Proposed Method: ADAPTIVE

The goal of the proposed scheme is to improve QoS as the system utilization while reducing the number of mode switches (MS_{HC}) at run-time. The C^{LO} values of HC tasks have a crucial role in improving the system objectives. Therefore it is a challenge to set C^{LO} for each HC task to draw a trade-off between the objectives: system utilization and the number of mode switches. To address the challenge, we monitor the run-time execution times of HC tasks and adapt their C^{LO} at run-time to achieve a higher QoS while having fewer mode

switches based on the variation in execution times due to the input and environmental changes. Fig. 2 shows an overview of *ADAPTIVE*, which consists of design- and run-time phases. Here, the task schedulability must be guaranteed at both phases, and the C^{LO} adaptation is done at run-time. In the following, we explain them in detail.

A. Design-Time Exploration

To analyze and schedule the HC and LC tasks in the system, first, the WCETs, required by the tasks must be obtained. Here, the C^{HI} of HC tasks are computed by using the OTAWA tool [8], which provides a safe and conservative execution time-bound. The WCETs of LC tasks can also be determined by using the OTAWA. In addition, to obtain the initial C^{LO}, we run the benchmarks with various data set inputs and set the maximum value of these actual execution times, as C^{LO} for each HC task. These analyzed data have been used to check the task schedulability by the Utility Checker Unit, which is shown in the design-time phase of Fig. 2.

In this paper, the existing MC scheduling technique, EDF-VD [1], [3], [5] (which has been used in many studies in the last decade), is applied. However, the proposed scheme is applicable to any scheduling algorithm. If U_l^k denotes total utilization of tasks with the same criticality level l ($l \in \{LC, HC\}$) in the mode k ($k \in \{LO, HI\}$), where $U_l^k = \sum_{i \in \{LC,HC\}} \frac{C_i^k}{T_i}$, Eq. (1) must be satisfied to guarantee schedulability by EDF-VD. This equation presents the necessary and sufficient conditions to guarantee the task schedulability in both LO and HI modes and meeting the deadlines, even if the system switches to the HI mode [5], [14]. The utilization (UMC) which is the maximum value of two phrases shown in Eq. (1), must be always less than one in EDF-VD, which is checked by Utility Checker Unit.

$$U_{HC}^{LO} + U_{LC}^{LO} \leq 1 \;\; \& \;\; U_{HC}^{HI} + U_{LC}^{HI} + \frac{U_{HC}^{LO} \times (U_{LC}^{LO} - U_{LC}^{HI})}{1 - U_{LC}^{LO}} \leq 1 \tag{1}$$

B. Run-Time Adaptation

The crucial research questions that should be addressed in the run-time phase are: 1) How to vary C^{LO} of HC tasks with no adverse effect on meeting the other tasks' deadlines, 2) How to design a scheme for determining the C^{LO} at run-time, to not be affected and varied by sudden changes of execution times, 3) How to design a scheme with low timing overheads during run-time to have no impact on task scheduling and deadline misses, 4) What are the best C^{LO} for the tasks to effectively keep the system away from mode switching while having high processor utilization and consequently, a high QoS value. Following the above questions, machine learning techniques can effectively help to design an adaptive MC system to make a reasonable trade-off between the objectives according to environmental changes (i.e., input values variation).

At run-time, MC tasks are executed on the platform, controlled by MC Task Scheduler Unit on the OS, shown in Fig. 2. The system monitors the tasks from two aspects:

1) Each task execution finishes or not: The actual execution times are stored in the case of complete execution. Besides, in the case of task overrun, the system switches to HI mode, and MC Task Scheduler Unit executes HC tasks by considering their C^{HI} and LC tasks with their QoS^{min}. The Processor Queue Checker Unit keeps track of the processor queue when the system can switch back to LO mode.
2) The system reaches the task set hyper-period or not: At the end of each hyper-period, the agent starts its operation by employing the data like actual execution times, the number of mode switches, and the QoS of LC tasks in the last hyper-period. Based on these historical data, the agent outputs are the new C^{LO} values of HC tasks, used in the next hyper-period. Since the utilization of HC tasks in LO mode would be changed by updating C^{LO}s, the new virtual deadlines (to be used in the EDF-VD algorithm) are determined by the Virtual-Deadline Update Unit.

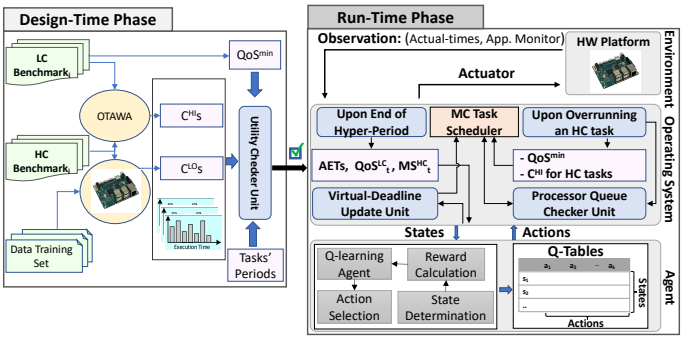


Fig. 2: An overview of design-time and run-time phases in ADAPTIVE.

Hence, the learning process is separate from the task scheduler, and we do not use learning techniques for task scheduling. The EDF-VD schedulability formulae are checked for each WCET change (at the end of each hyper-period). Although this time is in the order of μSeconds and can be negligible, we count this time as part of learning time. This timing overhead is considered as a task with the WCET equal to the maximum timing overhead to ensure it does not impact other tasks' deadlines. This learning overhead is reported in Section V-A. We describe below how the agent is designed to update the C^{LO}.

*1) **Learning-Based System Properties Improvement**:* Reinforcement Learning (RL) could be applied to systems with considerable dynamism through trial-and-error. By using historical data and learning from past events, it can improve performance based on dynamic changes [15]. The Q-learning/SARSA technique, which is recently used in many applications, such as robotics, and Unmanned Aerial Vehicles, uses the RL to perform the run-time management of the system properties. This technique is a value-based algorithm that iteratively collects the current system state and determines the next action to change the state. The process is repeated until meeting the predefined criterion or objectives are no longer significantly improved.

RL technique consists of the three main components [15]: 1) a discrete set of States = $\{s_1, s_2, ..., s_l\}$, 2) a discrete set of Actions = $\{a_1, a_2, ..., a_k\}$, and 3) reward function R^{MC}. To reach the favorable reward, the technique learns a lookup table (i.e., Q-table) with (s_t, a_t) pairs ($a_t \in$ Actions and $s_t \in$ States). The states and actions determine the rows and columns of the Q-table of the learning-based algorithm, respectively (shown in Fig. 2). As mentioned, a value-based algorithm is utilized, represented with $Q(s_t, a_t)$ in the Q-table, and determines the quality of the action taken at the particular state. In every iteration, the Q-values are updated based on the corresponding computed reward according to Eq. 2, which is based on the SARSA technique, one of the RL methods for objective improvement [16].

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(R^{MC} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \tag{2}$$

s_t, and a_t represent the state and action of the system at time t, respectively. s_{t+1} and a_{t+1} also indicate their values at time $t+1$. The α ($0 < \alpha \leq 1$) determines the learning rate of overriding the old data in the table by the new acquired data. R^{MC} is the

reward function, and γ is the discount rate to determine the importance of the future reward ($0 < \gamma < 1$).

State Determination: There are various criteria for determining the system states. In *ADAPTIVE*, the system states (i.e., the rows of the Q-table) indicate the rate of LC tasks' execution, i.e., the tasks' periods at run-time, to the minimum tolerable period in LC tasks. We define 10 ranges to determine the rate of LC tasks' execution. As a result, States=$\{0.1, 0.2, ..., 1\}$.

Learning Action Determination: In this paper, the well-known ϵ-greedy policy (a method for determining the optimal action) has been exploited, where a random action is selected from the action set with the probability of ϵ, i.e., the best action is selected with the largest Q-value with the probability of $1 - \epsilon$. We first use a dynamic ϵ-greedy policy [17] with the maximum value of 0.5 to prevent the probability of learning being stuck at a few Q-values. Afterward, the fixed ϵ-greedy policy is used to ensure that the system reaches the optimum state and chooses the best action based on the Q-values. We have assumed k actions, where the action space in the Q-table illustrates an increase and/or decrease in C_i^{LO} of some/all HC tasks according to a coefficient of *WCET*'s prediction accuracy. To limit the number of feasible actions and reduce the complexity and convergence issues, we have considered three scenarios of increase ($WCET_{NumT}^{inc}$), decrease ($WCET_{NumT}^{dec}$), and increase-decrease ($WCET_{NumT}^{inc,dec}$) ($opr = \{inc, dec, inc/dec\}$). $WCET_{NumT}^{inc}$ ($WCET_{NumT}^{dec}$) shows an increase (decrease) in C_i^{LO} of $NumT$ HC tasks, where the value of $NumT$ can be $\{1, 2, ..., n_{HC}\}$ (n_{HC} is the total number of HC tasks). $WCET_{NumT}^{inc/dec}$ presents that the C_i^{LO} for half of HC tasks is increased and for others decreased. In fact, in this scenario, $max(NumT)=\frac{n_{HC}}{2}$. Thus, there are $k = 2.5 \times n_{HC}$ actions in the system. The step of increase/decrease in C_i^{LO} is determined based on a coefficient of *WCET*'s prediction accuracy.

$$\text{Actions} = \{WCET_{NumT}^{opr}\} \quad opr \in \{inc, dec, inc/dec\} \tag{3}$$

To select the tasks to do the actions, we first sort the tasks in increasing order of $C_i^{LO} - AET_i$ values in the last hyper-period (AET_i is the actual execution time), then the increase (decrease) action applies to the $NumT$ tasks with smaller (greater) $C_i^{LO} - AET_i$. Since a task may release several times in a hyper-period and the actual execution time is different in each release, we have to predict the actual execution time according to the previous task's execution times. This prediction is based on Eq. 4, where $AET_i(t+1)$ is the predicted execution time of task τ_i, rc_i is regression coefficient, and er is the error. For evaluations, x is assumed to be eight due to various experiments that we performed to achieve lower prediction error with no timing overhead that can impact tasks' timeliness. For example, for a task, (x,er,time[$\mu Second$])=(2,0.110,0.86), (4,0.094,1.12), (8,0.077,1.59), (10,0.071,1.92). Since the error (er) does not change much from 8 to 10, compared to 4 to 8, x= 8 is a good value with less timing overhead.

$$AET_i(t+1) = \sum_{k=0}^{x} AET_i(t-k) \times rc_k + er \tag{4}$$

Reward Computation: The reward indicates how well the learning procedure has performed in the previous step. We calculate the reward at the end of each hyper-period. The number of mode switches should be reduced while increasing the number of scheduled LC tasks to improve the QoS. Eq. (5) shows the reward function based on these objectives. MS_{HC} indicates the mode switches (computed by the number of overrun HC tasks), and β_1 and β_2 are constants, and set to 0.5 ($\beta_1+\beta_2$ = 1) in this work.

$$R^{MC} = \beta_1 \times MS_{HC} + \beta_2 \times \text{QoS} \tag{5}$$

To compute the number of mode switches, Eq. (6) considers three scenarios. If the percentage of overrun HC tasks falls into the unsafe zone that may cause frequent mode switches, the decision is penalized. Accordingly, it results in a negative value for the reward function, which decreases the Q-value in Eq. (2). Eq. (7) shows how to compute the percentage of overrun tasks.

$$MS_{HC} = \begin{cases} +\Gamma(-\Gamma) & PT_{HC}^{ovr} = 0 (PT_{HC}^{ovr} = 1) \\ 1 - \frac{1}{10 \times (1 - PT_{HC}^{ovr})} & 0 < PT_{HC}^{ovr} < 1 \end{cases} \tag{6}$$

where $\Gamma > 0$, and has a constant value (set to one in this work).

$$PT_{HC}^{ovr} = \frac{\#\text{HC-Tasks}|_{\frac{C_i^{LO} - AET_i}{T_i} < 0}}{n_{HC}} \tag{7}$$

n_{HC} is the number of HC tasks and AET_i is the actual execution time of τ_i during a hyper-period (computed by Eq. (4)).

*2) **Algorithm**:* Algorithm 1 presents the pseudo-code of *ADAPTIVE*, including the task scheduling and learning procedures. As inputs, tasks and the minimum QoS are taken. QoS improvement, analyzed C^{LO}s, and the scheduled tasks are outputs. At each time, the scheduler checks the task's status, whether they are released or overrun (results mode switching) (line 3). Tasks are scheduled based on the EDF-VD (line 4). In the case of mode switching to HI, the LC tasks are executed based on their minimum service requirements. Line 5 checks whether each task output is ready, and in the case of being ready, the task is removed from the core queue, and the value of QoS and PT_{HC}^{ovr} are updated (lines 6,7). The learning process is conducted at the end of each hyper period (lines 8-23). The number of scheduled LC tasks is used to determine the state (line 9). In $\epsilon-$Greedy policy, if a random number is less than ϵ, random action is selected (line 12, learning exploration phase); otherwise, the action with the maximum value in the Q-table is selected for that particular state (line 13,

Algorithm 1 Run-Time Adaptation Scheme

Input: Task Set, QoS^{Min}
Output: Scheduled Tasks, QoS, C_{HC}^{LO}s
1: **procedure** ADAPTIVE FUNCTION()
2: **for** t = 1 **to** *Time* **do**
3: [$Mode_{MS}^{Sys}$,ReadyTasks] = **TaskStatusCheck** (Tasks,$Mode_{MS}^{Sys}$)
4: [Sch_{tasks}] = **EDF-VD** (ReadyTasks)
5: $Flag_{output}$=**TaskOutputCheck**(Tasks)
6: **if** $Flag_{output}$ == 1 **then** Update QoS & PT_{HC}^{ovr};
7: **end if**
8: **if** mod(t,HP)==0 **then**
9: State= **Deter-State** (#Scheduled-LCTasks)
10: *k= rand (1);* //(0<k<1)
11: *//ϵ-Greedy Policy*
12: **if** $k < \epsilon$ **then** a_t = *argrand* (Actions)
13: **else** a_t = *argmax* (s_t, Actions)
14: **end if**
15: Set the new tasks' C^{LO}s based on the action
16: R^{MC} = **CompRward** (PT_{HC}^{ovr},QoS)//Eq.(5)
17: $Q(s_t, a_t) = Q(s_t, a_t) + \alpha(R^{MC} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$ //Eq. (2)
18: *UMC(t)*=CompUtil (Tasks)
19: **if** *UMC(t)>1* **then** //based on the new C^{LO}s
20: Tasks = Deter-ExeJobs (Tasks);
21: **end if**
22: Tasks = Deter-VirtualDeadline (Tasks);
23: **end if**
24: **end for**
25: **end procedure**

TABLE II: System performance at run-time for different scenarios

	$Avg(QoS)^*$	$Avg(\#MS^{Sys})^{**}$	$Util^{Wst}{}^{**}$	$max(U_{LC}^{LO})^*$
Liu'18 [5] $\lambda = \frac{1}{4}$	50.0%	0	43%	50%
Liu'18 [5] $\lambda = \frac{1}{8}$	49.3%	5.81	28%	65%
Ranjbar'21 [3]	58.1%	1.16	33%	63%
ADAPTIVE	68.9%	2.12	16%	58%

* Higher is better ** Lower is better

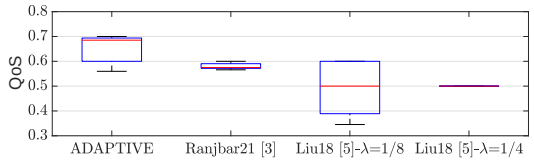


Fig. 3: Quality-of-Service (QoS) values during run-time for different scenarios

learning exploitation phase). Based on the chosen action, new C^{LO} values are determined for some HC tasks (line 15). Consequently, the reward function is used to update the Q-table (lines 16,17). In lines 18-22, the service adaptation and assigned virtual deadlines to HC tasks are computed based on the updated HC tasks' utilization to guarantee the schedulability Thus, the maximum amount possible of service adaptation is determined by finding the maximum execution rate of LC tasks, i.e., reducing their periods to release more often.

V. Experiments

In this section, the *ADAPTIVE* efficacy is evaluated in terms of learning overheads, mode switches, QoS, utilization waste.

A. Investigating Timing And Memory Overheads of ML Tech.

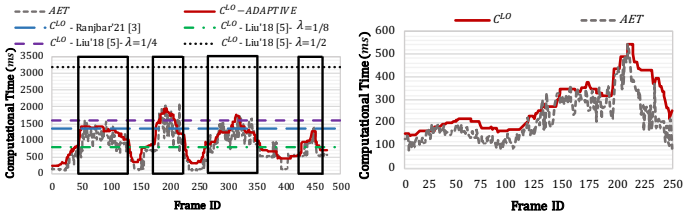
We first analyze the timing overhead of the learning process in each hyper-period on ODROID XU4, ARM A7, with 1.4GHz. Consider a system with n tasks, in which $n/2$ of them have HC. The timing overhead is different for the exploration and exploitation phases of the learning process. The $\epsilon-$Greedy policy makes a trade-off between the exploration and exploitation phases. We measured the learning time at run-time, and the average and maximum of exploration (exploitation) timing overhead in ARM core are $19\mu s$ ($52\mu s$) and $2.11ms$ ($4.15ms$), respectively. As a result, since the maximum timing overheads are almost significant for real-time systems, we can consider the learning process as a task with the WCET, equal to the maximum learning timing overhead, and a period equal to hyper-period, while checking the schedulability at design-time. As a result, it can guarantee that the tasks' timeliness is maintained at run-time. Besides, from the memory overhead perspective, we need to clarify the required memory space for storing the Q-table. We store a two-dimensional array with size (*State*) rows and size (*Action*) columns. Since the value of a table cell is a float number in the range of [-2,2], at most 32 bits are required for storing each cell. Thus, we need size(*State*)×size(*Action*)× 32 bits to store the Q-table. For an application with 40 tasks, the amount of required memory space for saving the Q-table with ten states would be 10× 2.5×(40/2)×32 bits= 16 KB.

B. Evaluation With Real Application Model

We conducted experiments by various real benchmarks from MiBench benchmark suite [18], like automotive, and network. In this experiment, ‹insert-sort›, ‹qsort›, ‹bitcount›, ‹dijkstra›, and ‹FFT› are considered as HC tasks, and ‹corner›, ‹edge›, ‹smooth›, ‹epic›, and ‹matrix-mult› as LC tasks. To obtain their execution times, we run the benchmarks with various inputs on Cortex A7 of the ODROID XU4 board (equipped with Ubuntu 18.04 as OS) with maximum frequency ($1.4GHz$). More detail on WCET values has been reported in [19]. We compare the results with the results of [3], [5]. As mentioned in

TABLE III: Number of deadline misses and gained utilization of different methods for Object Detection function in Fig. 4a, where there are many objects

Metrics	ADAPTIVE	[3]	[5] $\lambda = \frac{1}{2}$	[5] $\lambda = \frac{1}{4}$	[5] $\lambda = \frac{1}{8}$
$\#MS^{Sys}$	17%	11%	0	5%	45%
$Util^{Wst}$	28%	46%	76%	52%	47%



(a) Input video with many objects (b) Input video with few objects

Fig. 4: Learning process in adjusting C^{LO} for two video inputs of Object Detection function, compared to other methods in adjusting C^{LO}

[3], since most papers like [1], [5], [6], consider the same policy to determine the C^{LO} (i.e., defining a fraction of C^{HI} as C^{LO}), we select [5] as a representative of these schemes and do the experiments with two fractions of C^{HI} as C^{LO} ($\lambda = \frac{C^{LO}}{C^{HI}}$=[$\frac{1}{4}$, 1], [$\frac{1}{8}$, 1]). Besides, we investigate the system for 2000 hyper-periods of tasks. In the learning process, we set the values of γ to 0.2 and α to 0.5, which are determined based on a wide range of experiments that lead to the best improvement.

To evaluate different approaches in Table II, QoS represents the percentage of executed to total LC task instances during run-time ($QoS = n_{LC}^{schd}/n_{LC}^{max}$). MS^{Sys} indicates the number of mode switches per hyper-period, and $max(U_{LO}^{LC})$ represents the maximum processor utilization that can be assigned to LC tasks at design-time. $Util^{Wst}$ shows the average percentage of the difference between the WCET and actual execution times to WCET of tasks. As shown, *ADAPTIVE* can schedule more LC tasks; 10.8% and 19.3% more compared to [3], and [5] approaches, respectively. Although MS^{Sys} is more than the scenarios of [3], and [5] with $\lambda = \frac{1}{4}$, *ADAPTIVE* could overcome the significant performance loss due to executing more LC task instances in total. This can be achieved by determining the appropriate C^{LO}s during run-time, which is close to the actual execution times (17.7% closer on average, compared to [3], [5]). This fact can be observed with $Util^{Wst}$ values, compared to other approaches. In addition, although the scenario of [5] with $\lambda = \frac{1}{8}$ assigns more utilization to LC tasks at design-time, it has less QoS due to the more frequent mode switches. Although we assign an initial value of C^{LO} by running each benchmark on the platform and choosing the maximum of them (which causes lower utilization compared to [3], [5]), our learning approach is independent of how the C^{LO}s are set at design-time, and any of other design-time approaches, like the approach of [3] can be used.

Fig. 3 shows the variation of QoS values in different hyper-periods of the run-time phase. As shown, the scenario of [5] with $\lambda = \frac{1}{8}$ has a wide range of QoS values due to more mode switches. *ADAPTIVE* also represents a variation in the QoS values due to its adaptation to the input changes at run-time.

Now, we demonstrate the progress of the learning process in adjusting the C^{LO} during run-time for the two input videos from the object detection function, explained in the motivational example of Section I. Fig. 4 depicts the actual execution time trace and the adjusted C^{LO} for a time period during run-time for ADAPTIVE and the methods of [3], [5]. In ADAPTIVE, by changing the inputs which have low execution time values, the C^{LO} would be adjusted to the lower value intentionally. The C^{LO} is also re-adjusted when the value

(a) Margin threshold (b) Learning time (c) WCET Inc./Dec. steps

Fig. 5: Impacts of varying different parameters of learning process on QoS, mode switches and utilization waste.

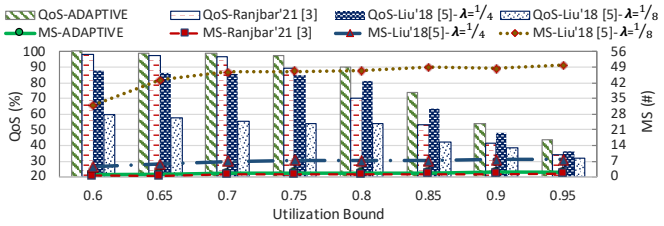


Fig. 6: QoS and mode switches in different approaches by varying utilization.

of execution times is increased. As shown in Fig. 4a, the methods of [3], [5] set the static C^{LO} which may lead to frequent task overruns (i.e., leads to regular mode switches) in the case that there are many objects to detect (lead to high computational time values) or poor utilization when there are few objects to detect in this function (like Fig. 4b). Although there are few errors while adjusting the C^{LO}, which leads to task overrunning and QoS degradation, the number of task overruns and the wasted processor utilization ($Util^{Wst}$) are less, and consequently, overall QoS value would be higher at the end of system execution. For example, in Fig. 4a, the black rectangles show the time periods in which some of the execution times are higher than C^{LO} for most methods, even though the execution times of these input videos are less than the adjusted C^{LO} in [5] with $\lambda = \frac{1}{2}$. Table III presents the percentage of task overruns (which leads to mode switches) and the average percentage of wasted utilization for ADAPTIVE and [3], [5]. Although the number of mode switches is higher than the results of some scenarios, the wasted utilization is lower compared to other methods, which leads to higher QoS (like what is discussed for Fig. 3 and Table II).

To evaluate *ADAPTIVE* in improving the QoS, we first analyze varying the margin threshold, which adjusts above the actual execution times to overcome high WCET reduction. Fig. 5a shows that the QoS improvement is less while the threshold is pessimistic, i.e., having a larger margin. In this case, fewer mode switches exist due to adjusting the WCETs so cautiously, and consequently, the utilization waste has a higher value. Based on this observation, having the margin threshold equal to 10% improves the QoS even with more mode switches. Now, we vary the time of training/exploring, during run-time, e.g., 400 or 1000 hyper-periods (HP) of total time (2000 HP) in Fig. 5b. Note that the learning process starts to learn for a period of time, and then the learned data is exploited for the rest of the time. Spending more time learning leads to more accurate results. Thus, the QoS is improved by 5.9% with increasing the learning time, with an insignificant increase in mode switches and utilization waste. To more accurately adjust the WCETs, we define steps for WCET increment/decrement at run-time. These steps are coefficients of the difference between the WCETs and actual execution times. Having a larger coefficient (0.1X to 0.5X) leads to adjusting faster to the actual execution times (i.e., having better QoS (7.95%) and less utilization waste (18%)), but it may cause more wrong decisions in learning (i.e., more mode switches). However, step=0.5X improves QoS more.

C. Evaluation With Synthetic Task Sets

We now carry out an extensive evaluation with the synthetic task sets to evaluate the *ADAPTIVE* efficacy, compared to [3], [5] in terms of varying utilization in Fig. 6. The synthetic task sets are generated for various utilization bounds (U_{bound}) in line with research works like [6], in the range of [0.60,1] with steps of 0.05. For each U_{bound}, 50 task sets are generated in which tasks' periods are selected in the range of [200, 1000]ms. Besides, balanced tasks are assumed in terms of criticality levels. The minimum QoS equals to 0.3 in this paper. According to Fig. 6, the ability to improve QoS is less by increasing the utilization bound due to having more HC and LC tasks in the system, and then, the probability of mode switches is increased. In [5], although considering a small value for λ like $\frac{1}{8}$, increases the assigned utilization to LC tasks, but it causes more mode switches and dropping more LC task instances, which leads to poor QoS. Besides, considering a large value ($\lambda = \frac{1}{4}$) decreases the utilization at design-time, but it increases the QoS due to fewer mode switches. The approach of [3] has better results in total in comparison with [5] due to making an ideal trade-off between the mode switches and utilization. Although [3] has a slight improvement in mode switches, compared to *ADAPTIVE*, our scheme can improve QoS more, due to considering the run-time behavior which causes different execution times.

VI. Conclusion and Future Work

This article proposed an adaptive scheme to analyze the MC tasks at run-time to determine their WCET based on the task behavioral changes. The proposed adaptive scheme employed the ML techniques to improve the QoS, while guaranteeing the task schedulability and timeliness. The proposed scheme improves the QoS for synthetic and embedded real-time benchmarks by 17.62% and 16.4% on average, respectively.

In future research, we would extend our scheme by reducing the complexity of the ML technique to reduce its timing overhead. Besides, a design-time data set training would be investigated to speed up the run-time learning process.

References

[1] S. Baruah *et al.*, "The Preemptive Uniprocessor Scheduling of Mixed-Criticality Implicit-Deadline Sporadic Task Systems," in *ECRTS*, 2012.

[2] A. Burns and R. I. Davis, "A survey of research into mixed criticality systems," *CSUR*, 2017.

[3] B. Ranjbar *et al.*, "Improving the timing behaviour of mixed-criticality systems using chebyshev's theorem," in *DATE*, 2021.

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

[5] D. Liu *et al.*, "Scheduling analysis of imprecise mixed-criticality real-time tasks," *TC*, 2018.

[6] H. Su *et al.*, "An elastic mixed-criticality task model and early-release edf scheduling algorithms," *TODAES*, 2016.

[7] R. Wilhelm *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *TECS*, 2008.

[8] C. Ballabriga *et al.*, "Otawa: An open toolbox for adaptive wcet analysis," in *SEUS*. Springer, 2010.

[9] B. Ranjbar *et al.*, "Motivating Agent-Based Learning For Bounding Time in Mixed-Criticality Systems," in *DATE*, 2023.

[10] D. Liu *et al.*, "Edf-vd scheduling of mixed-criticality systems with degraded quality guarantees," in *RTSS*, 2016.

[11] X. Gu and A. Easwaran, "Dynamic budget management and budget reclamation for mixed-criticality systems," *Real-Time Systems*, 2019.

[12] B. Hu *et al.*, "Ffob: Efficient online mode-switch procrastination in mixed-criticality systems," *Real-Time Systems*, 2019.

[13] B. Ranjbar *et al.*, "Learning-oriented qos- and drop-aware task scheduling for mixed-criticality systems," *Computers*, 2022.

[14] ——, "FANTOM: Fault Tolerant Task-Drop Aware Scheduling for Mixed-Criticality Systems," *IEEE Access*, 2020.

[15] S. Pagani *et al.*, "Machine learning for power, energy, and thermal management on multicore processors: A survey," *TCAD*, 2020.

[16] S. Dey *et al.*, "User interaction aware reinforcement learning for power and thermal efficiency of cpu-gpu mobile mpsocs," in *DATE*, 2020.

[17] D. Biswas *et al.*, "Machine learning for run-time energy optimisation in many-core systems," in *DATE*, 2017.

[18] M. R. Guthaus *et al.*, "Mibench: A free, commercially representative embedded benchmark suite," in *WWC-4 (Cat. No.01EX538)*, 2001.

[19] B. Ranjbar *et al.*, "BOT-MICS: Bounding time using analytics in mixed-criticality systems," *TCAD*, 2022.