Flexible Mixed-Criticality Scheduling with Dynamic Slack Management*

Xinyang Dong§
School of Computer Science and Engineering
Northeastern University, No. 195 Chuangzin Road
Hunnan District, Shenyang, Liaoning, P. R. China
dongxinyang@stumail.neu.edu.cn

Gang Chen
School of Computer Science and Engineering
Sun Yat-Sen University, No. 132 Waihuan East Road
Panyu District, Guangzhou, Guangdong, P. R. China
cheng83@mail.sysu.edu.cn

Mingsong Lv† and Weiguang Pang‡
School of Computer Science and Engineering
Northeastern University, No. 195 Chuangzin Road
Hunnan District, Shenyang, Liaoning, P. R. China
†lumingsong@cse.neu.edu.cn
‡1810591@stu.mail.neu.edu.cn

Wang Yi
School of Computer Science and Engineering
Northeastern University, No. 195 Chuangzin Road
Hunnan District, Shenyang, Liaoning, P. R. China

Department of Information Technology
Uppsala University, S-75105 Uppsala, Sweden
wang.yi@it.uu.se

Received 19 August 2019
Accepted 10 July 2021
Published 7 September 2021

*This paper was recommended by Regional Editor Tongquan Wei.
§Corresponding author.

This is an Open Access article published by World Scientific Publishing Company. It is distributed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 (CC BY-NC-ND) License which permits use, distribution and reproduction, provided that the original work is properly cited, the use is non-commercial and no modifications or adaptations are made.
Mixed-criticality (MC) systems have become hot topics in real-time embedded system design. A lot of research works have been proposed on modeling and scheduling MC systems since the first work presented by Vestal in 2007. (See Ref. 2 for a comprehensive survey of MC scheduling). An MC system aims to implement functionalities of different degrees of importance (so-called criticalities) on a common platform, so that higher important functionalities could be ensured with a higher criticality assigned.

In MC systems, tasks are usually distinguished by high-criticality tasks and low-criticality tasks. Each high-criticality task may have two worst-case execution time (WCETs) specifications in its task parameters: a low-criticality WCET and a high-criticality WCET. When the system starts, a high-criticality task is supposed to finish execution within the low-criticality WCET (a more optimistic estimation). It is possible that a high-criticality task may overrun its low-criticality WCET, which means the task needs more time to finish. Once the overrun of a high-criticality task happens, the system’s execution is switched from low-criticality mode to high-criticality mode. In classic MC models, the overrun of one high-criticality task will not cause other high-criticality tasks to enter their high-criticality behaviors together. The system will schedule computation resources according to this pessimistic assumption. Since more execution resources are provisioned for high-criticality tasks, more low-criticality tasks should be abandoned to compensate for such unnecessary resource overbooking during mode-switches. Therefore, such a mode-switch scheme is very pessimistic and impractical in terms of system resource utilization. Recently, a flexible mixed-criticality (FMC) model was proposed to address the problem. In FMC, the overruns of high-criticality tasks are independent. The overrun of one high-criticality task will not cause other high-criticality tasks to enter their high-criticality modes. And low-criticality tasks can be provided with service degradation dynamically according to the overruns of high-criticality tasks instead of being dropped directly. Therefore, FMC can avoid resource overbooking.

Keywords: Dynamic slack management; flexible mixed-criticality system; utilization-based analysis.

1. Introduction

Mixed-criticality (MC) system has attracted a lot of research attention in the past few years for its resource efficiency. Recent work attempted to provide a new MC model, the so-called Flexible Mixed-Criticality (FMC) task model, to relax the pessimistic assumptions in classic MC scheduling. However, in FMC, the behavior of MC tasks is still analyzed in offline stage. The run-time behavior such as dynamic slack has not yet been studied in FMC scheduling framework. In this paper, we present a utilization-based slack scheduling framework for FMC tasks. In particular, we monitor task execution on run time and collect dynamic slacks generated by task early completion. And these slacks can then be used either by high-criticality tasks to reduce mode-switches, or by low-criticality tasks so that less suspensions are triggered with more execution time, and thus quality of service is improved. We evaluate our approach with extensive simulations, and experiment results demonstrate the effectiveness of the proposed approaches.
However, the FMC model is still pessimistic, because each task is assumed to consume its WCET in the analysis, yet in real practice, tasks will most likely finish before their WCETs and therefore generate an amount of time slacks.

Since the classic MC models\(^3\)–\(^6\) adopt coarse-grained resource overbooking schemes mentioned above, the slacks collected during run time are usually tiny piece of CPU times which may not be large enough to compensate for such resource overbooking, which cannot be efficiently employed during run time. In contrast, FMC provides a more fine-grained mode-switch scheme where unnecessary resource overbooking will be avoided. Slacks generated during run time can be accumulated and provided for execution of the overrun task rather than degrading low-criticality tasks. Therefore, it is nontrivial to analyze the dynamic behavior of FMC model and manage the slacks because the slacks can be exploited much more high-level efficiently. Both high-criticality and low-criticality tasks can benefit from these time slacks. When a high-criticality task overruns its low-criticality WCET, the task can continue its execution by using the slacks without immediately triggering a mode-switch. Thus, system mode-switches can be postponed or even removed depending on how much free slack time can be used. For a low-criticality task, it can also use the slacks when it overruns its time budget, and thus its suspension can be postponed. Motivated by this idea, we proposed an FMC-based dynamic slack management framework, called FMC-DS, to improve the resource utilization by using dynamic slacks. The detailed contributions of this paper are as follows.

- We developed a light-weight slack management protocol that efficiently monitors and manages dynamic slacks which may be generated during run time. By modeling dynamic slacks as temporary (or aperiodic) real-time jobs (called TP-jobs), the slack management problem can be transformed into a real-time scheduling problem.
- We developed an extended mode-switch scheme to efficiently utilize the remaining slacks under FMC scheduling framework. During run time, dynamic slacks are accumulated for use to postpone system mode-switch and reduce the suspension of low-criticality tasks.
- Experimental results show that FMC-DS outperforms the existing static mode-switch scheme: FMC,\(^7\) classic EDF-VD,\(^6\) IMC\(^8\) in terms of reducing the suspension of low-criticality tasks and less system mode-switches with FMC scheduling guarantee.

The remainder of this paper is structured as follows. Related work is summarized in Sec. 2. Section 3 presents the system model and problem definition. An overview of our proposed techniques is given in Sec. 4. Experimental results are presented in Secs. 5 and 6 concludes the paper.

2. Related Work

The MC scheduling problems have been extensively explored and studied in different aspects during the past few years. In classical MC models,\(^3\)–\(^6\) a pessimistic mode-
switch scheme was adopted to ensure the safety of high-criticality task: whenever a
high-criticality task overruns, all low-criticality tasks are immediately abandoned
and all other high-criticality tasks are assumed to exhibit high-criticality behaviors
together. Most recently, solutions7,9–12 have been proposed to address the problem.
Santy et al.9 proposed an approach that low-criticality tasks will get resource until
the high-criticality tasks are guaranteed to have enough execution time. Fleming and
Burns10 introduced the notion of importance to decide which tasks are suspended
first. Chen et al.7 proposed a FMC model with an independent mode-switch scheme
for high-criticality tasks and a dynamic degradation scheme for low-criticality tasks.
Lee et al.11 proposed an MC-ADAPT framework supporting online adaptive task
dropping under an independent mode-switch scheme. Guo et al.12 proposed a new
model to provide a graceful degradation of service for low-criticality tasks by re-
defining the high-criticality mode. Cao et al.13 proposed a method to maximize the
lifetime of MC embedded systems considering the transient fault tolerance of system.
However, those works are all focusing on analyzing the static scheduling perfor-
ance. The run-time behaviors such as dynamic slacks have not been involved.

Little work has been conducted on analyzing run-time behavior in MC systems.
Hofmann et al.14 used slack to improve the end-to-end latencies and average latencies
of Ethernet Networks. Su et al.15,16 exploited dynamic slack to provide minimal service
guarantee for low-criticality tasks. However, this work is based on classic MC task
models5 which applies impractical assumptions. Recently, Hu et al.17 applied a similar
approach in Ref. 18 to adaptively detect available slack and further use the slacks to
handle overruns during run time. However, the analysis approach for detecting run-
time slack in Refs. 17 and 18 relies on real-time calculus theory,19 which is not suitable
for run-time analysis due to large analysis overhead. Reference 20 extended Hu’s
work17 towards arbitrary activation tasks and multiprocessor. Reference 21 proposed
an approach to generate the slacks after the task has finished. However, the analysis
approaches used by these works have still not been improved.

In contrast to the above works, we proposed a light-weight utilization-based slack
management technique under the FMC scheduling framework. FMC provides a more
fine-grained transition scheme,7 which can analyze and exploit dynamic slacks more
efficiently. Furthermore, dynamic slacks are modeled as temporary (or aperiodic)
real-time jobs (called TP-jobs), which can be seamlessly integrated into task
scheduling in a light-weight manner.

3. System Model and Problem Description

3.1. Flexible mixed-criticality task model

In this paper, we study slack scheduling problem under the FMC scheduling frame-
work. According to Ref. 7, we adopt FMC model with the following assumptions. We
consider generalized MC uniprocessor systems with a task set \( \gamma \) that consists a finite
collection \( \{ \tau_1, \tau_2, \ldots, \tau_n \} \) of \( n \) independent sporadic tasks. Each task \( \tau_i \) in \( \gamma \) generates
an infinite sequence of jobs and can be specified by a tuple \( \{T_i, D_i, L_i, C_i\} \), where

- \( T_i \) is the minimum job-arrival intervals.
- \( D_i \) is the relative deadline. In this paper, we consider an implicit-deadline task set with \( T_i = D_i \) for every task \( \tau_i \).
- each task is either a low-criticality task or a high-criticality task.
- \( L_i \in \{\text{LO, HI}\} \) denotes the criticality level of task \( \tau_i \).
- \( C_i = \{C_i^{\text{HI}}, C_i^{\text{LO}}\} \), where \( C_i^{\text{LO}} \) and \( C_i^{\text{HI}} \) denote the low-criticality and high-criticality WCET of high-criticality task \( \tau_i \), respectively.

In contrast to the classical MC models, FMC model allows high-criticality tasks to trigger mode-switch independently, which means the overrun of one high-criticality task triggers only itself into high-criticality mode rather than all high-criticality tasks. And the quality of service of low-criticality tasks is dynamically decided by the overruns of tasks. Consequently, the criticality mode of the system in FMC is specified by the number of high-criticality tasks that overrun, i.e., a task set consists of \( k \) high-criticality tasks, and at most \( k + 1 \) criticality modes of the system will be redefined. Therefore, we introduce the following definition.

**Definition 1 (k-level high-criticality mode).** At a given instant of time, if \( k \) high-criticality tasks have entered high-criticality mode, then the system is in \( k \)-level high-criticality mode. For low-criticality mode, we define that the system is in 0-level high-criticality mode.

When \( k \) mode-switches have occurred, the run-time budget for a low-criticality task will be decreased to make room for mode-switched high-criticality tasks. This is implemented by multiplying a factor \( z_j^k \) ranging from 0 to 1 to the original WCETs of all low-criticality tasks. \( z_j^k \) can be computed during run time based on how much extra execution time the overrun high-criticality tasks need. The detailed definition is as follows.

**Definition 2 (Service level of low-criticality task).** When the system has undergone \( k \) mode-switches, the budget of low-criticality task \( \tau_j \) is degraded to \( z_j^k \cdot C_j^{\text{LO}} \) in one period \( T_j \), where \( z_j^k \) denotes the service level of \( \tau_j \) when \( k \) mode-switches have occurred. Specially, we define the service level \( z_0^0 = 1 \), when \( \tau_j \) is in low-criticality mode.

**Execution Semantics of FMC:** In FMC model, the system starts with low-criticality mode (0-level high-criticality mode), where all tasks are scheduled with their low-criticality WCETs. An overrun only triggers the task itself to enter its high-criticality mode, while all other high-criticality tasks remain at their previous criticality levels. The degradation of low-criticality tasks only needs to compensate the resource overbooking of a single mode-switched task. In particular,
for the $k$-level high-criticality mode, the execution budgets of low-criticality tasks are dynamically degraded to $z_j^k \cdot C_{LO}^j$. If any low-criticality task $\tau_j$ overruns its degraded budgets (has used up its execution budget) in the current period, it will be suspended immediately.

3.2. Problem description

This paper considers the analysis and management of dynamic slacks for FMC scheduling at run time. The dynamic slacks generated from early completion of tasks will bring more free budgets to compensate for the overruns of high-criticality tasks. We consider a new mode-switch scheme based on FMC, in which the overrun tasks will not immediately book overrun budgets and enter into high-criticality mode when an overrun occurs. Instead, the dynamic slacks will be consumed by such tasks to complete their overrun executions. Consequently, the overrun tasks may complete their executions with the resource provided by the slacks before triggering mode-switch of system. So, in this paper, we aim to solve the following key problems:

1. how to analyze the online behavior of FMC tasks and collect the dynamic slacks during system execution.
2. how to manage (i.e., consume and update) slacks to handle overrun during run time.
3. how to find a more flexible and resource-efficient mode-switch scheme based on FMC to improve quality of service of low-criticality tasks while guaranteeing the MC schedulability.

Now, we can define the problem that we studied as follows:

**Given an MC tasks set $\gamma$ based on FMC scheduling, we are to find a light-weight management strategy to collect and manage the dynamic slacks to maximize the quality of service of low-criticality tasks, while keeping the schedulability of all the tasks.**

4. Flexible Mixed-Criticality Scheduling with Dynamic Slack

In this section, we propose an FMC-based dynamic slack management framework (called FMC-DS), which dynamically manages system slacks and adaptively postpones system mode-switches. We first give an overview of the FMC-DS framework. Then we explain our Dynamic Slack Management Protocol (DSMP) used to manage the dynamic slacks. At last, we give the scheduling algorithm and analyze the complexity of the algorithm.

4.1. The overview of FMC-DS

In this paper, we aim to utilize the dynamic slacks to compensate the overruns in FMC, such that tasks can commit their execution behaviors in the previous mode by postponing mode-switches for high-criticality tasks or reducing suspensions for
low-criticality tasks. In FMC-DS, we introduce a system state called \textit{intermediate mode} to model such system behavior. The \textit{intermediate mode} is inserted before each mode-switch, where the dynamic slacks are utilized to compensate the resource overbooking caused by overruns. Therefore, the number of \textit{intermediate modes} is specified by the number of high-criticality modes of FMC. The system state of \textit{intermediate mode} can be defined as follows.

\textbf{Definition 3 (k-level intermediate mode).} When the system is in \((k-1)\)-level high-criticality mode \((k > 0)\) and any task \(\tau_i\) overruns, the system will enter the \(k\)-level intermediate mode. In the \(k\)-level intermediate mode, the execution behaviors of all tasks are the same as in \((k-1)\)-level high-criticality mode.

Based on this extension, the system semantics are further extended as follows: the system initially goes through low-criticality mode (0-level high-criticality mode), then followed by 1-level intermediate mode and 1-level high-criticality mode successively, and consequently undergoes among \(k\)-level intermediate mode and \(k\)-level high-criticality mode, as it is shown in Fig. 1. For ease of presentation, we focus on the system behavior of FMS-DS in this section. We assume slacks are collected on the

Fig. 1. An overview of FMC-DS.
event of task completion and updated by the scheduler. The details of slack management schemes are further introduced in Sec. 4.2. The semantics of FMS-DS is detailed as follows:

(1) **The low-criticality mode (0-level high-criticality mode).** All tasks start in this mode. As long as no task violates its low-criticality WCET, the system remains in this mode. All the tasks are scheduled with $C_i^{LO}$ and are guaranteed to meet their deadlines. The slacks will be detected during run time and managed by the scheduler (Sec. 4.2).

(2) **Transition(1).** If any high-criticality task $\tau_i$ overruns its $C_i^{LO}$, the system goes to 1-level intermediate mode.

(3) **$k$-level intermediate mode ($k > 0$).** Only the overrunning task (has triggered the transition process) will be executed in this mode. The accumulated slacks of system are consumed for this task to continue its remaining execution.

(4) **Switch.** When all the slacks are used up and the overrunning task still does not complete its execution, if this overrunning task is a high-criticality task, a mode-switch occurs and the system goes to the $k$-level high-criticality mode.

(5) **Recover.** (1) If the overrunning task has finished before all the slacks are used up, the system returns to the previous mode ($(k-1)$-level high-criticality mode).

(2) When all the slacks are used up and the overrunning task still does not complete its execution, if this overrunning task is a low-criticality task, it is suspended immediately and the system returns to the previous mode.

(6) **$(k-1)$-level high-criticality mode ($k > 1$).** In this mode, every high-criticality task in low-criticality mode $\tau_i$ will be scheduled with its low-criticality WCET $C_i^{LO}$; every high-criticality task in high-criticality mode $\tau_k$ will be scheduled with its high-criticality WCET $C_k^{HI}$; every low-criticality task $\tau_j$ is scheduled with its degraded (or reduced) WCET $z_j^{k-1} \cdot C_j^{LO}$. All the high-criticality tasks are guaranteed to meet their deadlines and the low-criticality tasks are treated by reducing their service budgets. The slacks will be detected during run time and managed by the scheduler (Sec. 4.2).

(7) **Transition(2).** If any high-criticality task $\tau_i$ in low-criticality mode overruns its low-criticality WCET $C_i^{LO}$, the system goes to the $k$-level intermediate mode. Or if any low-criticality task $\tau_j$ overruns its degraded (or reduced) WCET $z_j^k \cdot C_j^{LO}$, the system goes to the $k$-level intermediate mode.

(8) **Return.** When the system detects an idle interval, the system will go back to the low-criticality mode.

The FMC-DS mode-switch scheme proposed by this paper allows the overrun tasks to continue their executions and the system will go to an intermediate mode to present such behaviors. Hence, mode-switches of system can be delayed or even avoided by using such slacks to provide resource for overrun tasks and compensate
for their resource booking. Next, we will study problem on how to effectively manage (collect, consume and update) the slacks during run time.

### 4.2. Dynamic slack management protocol (DSMP)

In this section, we propose a protocol for FMC-DS to manage dynamic slacks effectively during run time, which is termed as DSMP. In this protocol, we model a piece of slack as a temporary (or aperiodic) real-time job called TP-job. The TP-job is specified with two timing parameters \((l, d)\), where \(l\) represents the length (WCET) of TP-job and denotes the amount of slack generated; \(d\) is the absolute deadline of the slack which equals to the deadline of the task that generates this slack. A special queue called TP-queue is introduced to store the slacks (i.e., TP-job) during run time.

We now present a slack management protocol to deal with three important management processes: slack collection, slack consumption, and slack update during run time. These three processes are detailed as follows:

**Process 1 (Slack Collection).** New slacks are generated due to early completion of job \(J_i\) of task \(\tau_i\). During run time, we monitor the event of job completion to detect slack. When the event of early completion is detected, we create a TP-job for this detected slack and insert it into the TP-queue. The timing parameters of the newly detected TP-job \(TPJ_i\) is inherited from the job \(J_i\) which generates this slack. Assuming that \(J_i\) with WCET \(C_i\) and absolute deadline \(\hat{D}_i\) completed with actual execution for \(E_i\) time units, the timing parameters of \(TPJ_i\) can be determined as a tuple \(((C_i - E_i), \hat{D}_i)\).

TP-jobs are organized in TP-queue in descending order by their deadlines. TP-jobs which have the same deadline will be merged into one big slack, as presented in the slack update process of TP-queue. Therefore, all deadlines of TP-jobs in TP-queue are different.

**Process 2 (Slack Consumption).** Once new slacks are detected, all the slacks in FMC-DS can be consumed by all tasks. The process of slack consumption is presented in Algorithm 2. In general, the slacks can be consumed in the following two scenarios:

- **Scenario 1: The system is not in intermediate mode.** In this scenario, we treat TP-job \(TPJ_i\) as a real-time job which is co-scheduled with other real-time jobs during run time. The execution time of \(TPJ_i\) can be consumed by currently-executing job \(J_j\) (the head of ready-task queue denoted as RT-queue), if the deadline \(\hat{D}_i\) of \(TPJ_i\) is less than the deadline \(D_j\) of real-time job \(J_j\). In this consumption scenario, currently-executing job \(J_j\) temporarily borrows free execution time from \(TPJ_i\), which will result in early completion of job \(J_j\). Correspondingly, the event of early completion of job \(J_j\) will generate a new slack job \(TPJ_j\), whose
timing parameters are inherited from $J_j$. The length of new generated slack depends on how much execution time borrowed from $TPJ_i$ during run time. (This process is illustrated in lines 3–9 of Algorithm 1.)

- **Scenario 2: The system is in intermediate mode.** In this scenario, the free slacks can be consumed by overrun $J_j$ to compensate the resource overbooking

---

**Algorithm 1. Slack Consumption**

```plaintext
Input: slack queue TP-queue, currently executing job $J_j$ of task $\tau_j$, intermediate mode indicator $\theta$, system time $t$, $J_j$’s real execution time $J_j.et$, $J_j$’s remain execution time $J_j.rem$

Output: TP-queue

1 if Isempty (TP-queue) then
2     return
3 $TPJ_i$ = Header (TP-queue)
4 if $\theta \equiv 0$ then
5     if $J_j.D_j > TPJ_i.\hat{D}_i$ then
6         if $J_j.et > TPJ_i.C_i$ then
7             Destroy ($TPJ_i$)
8             Enqueue ((Jj.Cj − Jj.et, Jj.Dj), TP-queue)
9             Enqueue ((TPJi,Ci,Jj.Dj),TP-queue)
10            SlackUpdate (TP-queue, t)
11       else
12         $TPJ_i.C_i$ = $TPJ_i.C_i$ − $J_j.et$
14         Enqueue ((Jj.et, Jj.Dj), TP-queue)
15         SlackUpdate (TP-queue, t)
16      else
17     for $TPJ_i \in TP-queue$ do
18         if $TPJ_i.C_i > J_j.rem$ then
19             $TPJ_i.C_i$ = $TPJ_i.C_i$ − $J_j.rem$
20         else
21             Destroy ($TPJ_i$)
22             $J_j.rem$ = $J_j.rem$ − $TPJ_i.C_i$
23             SlackUpdate (TP-queue, t)
24 if $J_j$ completes then
25     break
26 $\Theta = 0$
```

2150306-10
caused by overrun. To postpone mode-switches as late as possible, all slacks in \( TP-queue \) can be used until all slacks are used up (i.e., \( TP-queue \) is empty) or \( J_j \) completes. (This process is illustrated in lines 10–16 of Algorithm 1.)

**Process 3 (Slack Update).** During run-time, \( TP-queue \) is dynamically maintained according to the following rules.

- When a new slack is generated and a new \( TP-job \) is inserted into \( TP-queue \), \( TP-jobs \) in \( TP-queue \) are organized in descending order by their deadlines. The \( TP-jobs \) with the same deadlines will be merged (lines 8–10 of Algorithm 2).
- A consumed slack will be removed from \( TP-queue \) (lines 6 and 7 of Algorithm 2).
- For each \( TP-job \) in \( TP-queue \), if the length \( C_i \) of \( TPJ_i \) is larger than the time interval between current system time instant \( t_0 \) and its deadline \( \hat{D}_i \) (i.e., \( C_i > \hat{D}_i - t_0 \) holds), we say that \( TPJ_i \) is going to expire. \( TPJ_i \) will be updated from \((C_i, \hat{D}_i)\) to \((C_i - t_0, \hat{D}_i)\) (lines 2 and 3 of Algorithm 2). When the deadline of \( TPJ_i \) is larger than the current system time, \( TPJ_i \) will directly be removed from \( TP-queue \) (lines 4 and 5 of Algorithm 2).
- If the processor is idle, drop all the slacks and empty the \( TP-queue \).

### 4.3. An example of dynamic slack management protocol

Consider a task set with 3 periodic tasks \( \gamma = \tau_1, \tau_2, \tau_3 \) shown in Table 1. \( \tau_1 \) and \( \tau_2 \) are high-criticality tasks, and \( \tau_3 \) is a low-criticality task.
Figure 2 illustrates the execution process of the example. When the first instance of $\tau_1$ completes early at time 4, one unit of dynamic slack is generated since $\tau_1$'s WCET is 5. And a TP-job $S_1$ is created with (1,12) and inserted into the TP-queue. The system state is shown in Fig. 2(a).

The first instance of $\tau_2$ begins its execution and consumes $S_1$. At time 8, $\tau_2$ overruns its low-criticality WCET and the system goes into 1-level high-criticality mode. At time 13, $\tau_2$ completes. According to Process 2 of DSMP, $S_1$ will be updated to (1, 20). The system state is shown in Fig. 2(b).

![Figure 2: An example of DSMP.](image-url)
The first instance of $\tau_3$ begins its execution and completes at time 15 (assuming $\tau_3$ get two units of degraded (or reduced) budget after mode-switch). According to Process 2 of DSMP, since the deadline of $S_1$ (20) is no earlier than the deadline of $\tau_3$ (20), $S_1$ cannot be consumed by $\tau_3$. The system state is shown in Fig. 2(c).

The second instance of $\tau_1$ begins its execution and completes at time 20. According to Process 2 of DSMP, the slack will be consumed by $\tau_1$ and updated to (1, 27). The system state is shown in Fig. 2(d).

The second instance of $\tau_2$ begins its execution and consumes slack $S_1$. As $\tau_2$ is in its high-criticality mode, another two units of dynamic slack are generated when $\tau_2$ completes early at time 27. Then TP-job $S_2$ is created with (2, 40) and inserted into the TP-queue. Since $S_1$ is updated to (1, 40), according to Process 3 of DSMP, $S_1$ and $S_2$ will be merged to (3, 40). The system state is shown in Fig. 2(e).

The second instance of $\tau_3$ begins its execution with two units of degraded (or reduced) budget. According to Process 2 of DSMP, $S_1$ cannot be consumed by $\tau_3$. However, when $\tau_3$ overruns its (degraded) WCET at time 29. Then $S_1$ is consumed by $\tau_3$ for its extra execution. When it completes at time 30, $S_1$ is updated to (2, 40). The system state is shown in Fig. 2(f).

The third instance of $\tau_1$ begins its execution. When $\tau_1$ completes at time 35, $S_1$ will be updated to (2, 42). The system state is shown in Fig. 2(g).

During the time interval (35, 40), the CPU is idle and the system goes back to low-criticality mode. All the slacks are dropped according to Process 3 of DSMP. Then the third instance of $\tau_2$ begins its execution at time 40 and overruns its low-criticality WCET at time 44. Then $\tau_2$ triggers the system to enter 1-level high-criticality mode, and then early completes at time 45, which generates a TP-job $S_1$ (4, 60). After that, $\tau_3$ begins its execution and early completes at time 46, which generates a TP-job $S_2$ (1, 60). Then $S_1$ and $S_2$ will be merged into a new one (5, 60). The system state is shown in Fig. 2(h).

### 4.4. Implementation of FMC-DS

In this subsection, we provide a detailed description on how FMC-DS works during run time. The FMC-DS scheduler will run on periodic scheduling points, and the actually period is determined by system designers as long as it does not postpone task execution. At each scheduling point, jobs are selected for execution according to the following rules.

There is a criticality level indicator $\Gamma$, initialized to 0, and an indicator $\Theta$ to denote whether the system is in intermediate or not, initialized to 0;

**CASE 1:** $(\Gamma = 0$ and $\Theta = 0$)

(a) Suppose a job $J_i$ of a task $\tau_i \in \tau$ arrives at time $t$. If $\tau_i$ is a low-criticality task, $J_i$ will be assigned with absolute deadline and WCET equal to $(t + T_i)$ and $C_{i}^{LO}$, respectively; otherwise, $J_i$ is assigned with absolute deadline and WCET equal
to \((t + x \cdot T_i)\) and \(C_i^{LO}\), respectively.\(^a\) Then \(J_i\) is pushed into the ready queue to be scheduled. (we use \(\hat{D}_i\) to denote absolute deadline of \(J_i\) in the following description for simplicity).

(b) At each instant the waiting job with the earliest deadline is selected for execution.

(c) Suppose a job \(J_i\) of a task \(\tau_i \in \tau\) finishes at time \(t\). If the real execution time \(E_i\) of \(J_i\) is smaller than its low-criticality WCET \(C_i^{LO}\), \(J_i\) completes early. Then a new slack is generated with parameters \(((C_i^{LO} - E_i), \hat{D}_i)\) and stored in the slack queue.

(d) At each instant if the head (the slack with the earliest deadline of all) of the slack queue can be consumed by the currently executing task, the slack queue is updated accordingly.

(e) If the currently executing job executes for more than its low-criticality WCET without signaling completion, then the system enters into 1-level intermediate mode, and \(\Theta \rightarrow 1\).

CASE 2: \((\Gamma \equiv (k-1)/(k > 1) and \Theta \equiv 0)\)

(a) Suppose a job \(J_i\) of task \(\tau_i \in \tau\) arrives at time \(t\). If \(\tau_i\) is a high-criticality task in high-criticality mode, \(J_i\)'s scheduling absolute deadline is changed to \((t + T_i)\) and WCET is changed to \(C_i^{HI}\); if \(\tau_i\) is a high-criticality task in low-criticality mode, \(J_i\)'s scheduling deadline and WCET are assigned as its previous scheduling behavior; if \(\tau_i\) is a low-criticality task, \(J_i\) is provided with degraded service and its time budget is assigned as \(z_i^{k-1} \cdot C_i^{LO}\). Then \(J_i\) is pushed into the ready queue to be scheduled.

(b) At each instant the waiting job with the earliest deadline is selected for execution.

(c) Suppose a job \(J_i\) of a task \(\tau_i \in \tau\) finishes at time \(t\). If the real execution time \(E_i\) of \(J_i\) is smaller than its corresponding WCET \(C_i\), \(J_i\) completes early. Then a new slack is generated with parameters \(((C_i - E_i), \hat{D}_i)\) and stored in the slack queue.

(d) At each instant if the head (the slack with the earliest deadline of all) of the slack queue can be used by the currently executing task, the slack queue is updated accordingly.

(e) If the currently executing job of a high-criticality task in low-criticality mode executes for more than its low-criticality WCET without signaling completion, then the system enters into \(k\)-level intermediate mode and \(\Theta \rightarrow 1\).

(f) If the currently executing job of a low-criticality task executes for more than its degraded (or reduced) WCET without signaling completion, then the system enters into \(k\)-level intermediate mode and \(\Theta \rightarrow 1\).

\(^a\)\(x\) denotes the virtual deadline factor of task \(\tau_i\).
If the ready queue is empty, the system detects an idle interval and transits back into low-criticality mode, and $\Gamma \to 0$.

**CASE 3: ($\Theta = 1$)**

(a) At each instant if the slack queue is not empty, the overrunning job $J_i$ of a $\tau_i$ (has triggered the system to enter this mode) is selected for execution and the slacks are being consumed by $J_i$. The slack queue is updated accordingly.

(b) If $J_i$ finishes its execution before all slacks are used up, the system returns to its previous mode and $\Theta \to 0$.

(c) If the slack queue is empty: if $\tau_i$ is a high-criticality task, the behavior of the system is changed from $k$-level intermediate mode to $k$-level high-criticality mode and $\Gamma \to k$; otherwise, the system returns to its previous mode. And $\Theta \to 0$.

### 4.5. Run-time complexity for FMC-DS

For the FMC-based uni-processor real-time system alone, a scheduler is responsible for arrival, preemption and completion of jobs and system mode-switch behaviors. For FMC-DS, the additional operations are summarized as follows:

- Assuming the system consists $n$ tasks, upon arrival or completion of a job, FMC-DS updates slacks by Algorithm 2 with the worst-case time complexity of $\Omega(n)$ (traversing $TP$-$queue$ in time $\Omega(n)$).
- When the system does not stay in intermediate mode, FMC-DS allocates the slacks to jobs with the complexity of $\Omega(1)$.
- Whenever a job overruns and triggers the system into intermediate mode, FMC-DS allocates the slacks to job with the complexity of $\Omega(1)$.
- When the system stays in intermediate mode, and the slacks are not enough for handling overruns, a mode-switch occurs with the complexity of $\Omega(1)$.

Note that FMC can be implemented with the run-time complexity of $\Omega(n \cdot \log n)$.\(^7\) FMC-DS can implemented with the same time complexity of FMC, upon the same event: arrival, preemption and completion of jobs. And the additional run-time overhead of FMC-DS comes from the update of slacks, which requires $\Omega(n)$ at each event. Compared with the work,\(^7\) FMC-DS is a more lighter weight solution.

### 5. Experimental Evaluation

#### 5.1. Experiment setup

In this section, we evaluate the performance of the proposed approach with three existing utilization-based schemes, including FMC,\(^7\) IMC,\(^8\) classic EDF-VD.\(^6\) Our experiments were conducted on randomly-generated MC task systems. We adopt a
similar random task generator as in Refs. 6, 7 and 11. The parameters of task generation are as follows:

- The period $T_i$ of each task is an integer drawn uniformly at random in $[100, 1, 000]$.
- For each task $\tau_i$, low-criticality utilization $u_i^{LO}$ is a real number drawn at random in $[0.05, 0.15]$.
- $R_i$ denotes the ratio of $u_i^{HI}/u_i^{LO}$ for every high-criticality task, which is a real number drawn uniformly at random in $[1, 5]$.
- $p_{Cri}$ denotes the probability that a task $\tau_i$ is a high-criticality task, and we set it as 0.5.
- For high-criticality task $\tau_i$, we set its WCET by $C_i^{LO} = \lceil u_i^{LO} \cdot T_i \rceil$ and $C_i^{HI} = \lceil u_i^{LO} \cdot R_i \cdot T_i \rceil$. And for low-criticality task $\tau_j$, we set its WCET by $C_i^{LO} = \lceil u_i^{LO} \cdot T_i \rceil$.

Given the utilization bound $u_B$, one task is generated until the following conditions are both satisfied: (1) $u_B - 0.05 \leq \max\{u_i^{LO} + u_i^{LO}, u_i^{HI}\} \leq u_B$; (2) at least 3 high-criticality tasks have been generated. Each simulation runs for $10^6$ time units. During the simulation, we refer to Ref. 11 to set the overrun probability $OP_i$ for each high-criticality task $\tau_i$, which determines the probability that high-criticality task $\tau_i$ executes beyond its low-criticality WCET. We use the parameter $\alpha$ ($0 < \alpha < 1$) to control the slack generation of MC tasks. The actual execution time of MC tasks is generated as follows:

- For high-criticality tasks, if no overrun occurs, the actual execution time is randomly generated from $(\alpha \cdot C_i^{LO}, C_i^{LO})$. Otherwise, execution time is generated from $(C_i^{LO}, C_i^{HI})$.
- For low-criticality tasks, the actual execution time of low-criticality tasks $\tau_j$ is randomly generated between $(\alpha \cdot C_i^{LO}, C_i^{LO})$. Therefore, the smaller $\alpha$ is, the more slacks will be generated.

In our experiments, we adopt two criteria PFJ and MST to measure and compare the performance of our proposed approach and related approaches. PFJ refers to the percentage of finished jobs of low-criticality tasks, and MST refers to the number of mode-switches triggered by high-criticality tasks. To ensure fair comparison, we generate a job trace for each task set offline and apply the same trace to all compared schemes to obtain PFJ and MST during run time.

5.2. Results

We first show the effectiveness of FMC-DS compared with the state-of-the-art approaches. We vary utilization bounds $u_B$ from 0.7 to 0.95 with step size of 0.05,
and set the value of $\alpha$ as (the default value) 0.4 to evaluate online performance in terms of PFJ and MST. Figures 3 and 4 show the average PFJ and MST with varying utilization bounds $u_B$ for different overrun probability $\text{OP}_i \in \{0.1, 0.3, 0.5, 0.7\}$, respectively. The results show that FMC-DS consistently outperforms the other three schemes in terms of both PFJ and MST. This performance gain is achieved by the fact that FMC-DS provides a slack management mechanism that uses dynamic slacks to compensate for the execution of overrun tasks and adopts a more flexible and resource-efficient mode-switch scheme: the overrunning high-criticality tasks will not directly switch into high-criticality mode. Instead of directly switching into high-criticality mode, high-criticality tasks can utilize free slack to postpone mode-switches as late as possible. At the same time, FMC-DS allows low-criticality tasks themselves to continue execution instead of being suspended when they overrun their degraded budgets, which also improves PFJ.

Next, we will show how factor $\alpha$, which controls the amount of generated slack time, affects the online performance during run time. We set $u_B = 0.80$ and $\text{OP}_i = 0.5$ as default values. Figure 5 shows how the factor $\alpha$ affects the PFJ and MST. The $y$ axis shows the average PFJ and MST values, respectively, and the four different lines represent the four schemes’ behaviors with various $\alpha$ values. We observed the following trends: (1) FMC-DS outperforms all other three schemes in terms of PFJ and MST for all $\alpha$ values. (2) The performance of FMC-DS decreases in a faster trend compared to the compared schemes. The latter trend shows that FMC-DS is sensitive to the available slack time (note that a bigger $\alpha$ means less slack time) even
though FMC-DS still performs better than existing schemes since it can systematically manage run-time slacks and exploit slack usage as much as possible.

6. Conclusion

In this paper, we proposed the FMC-DS framework for MC systems. The framework pushes the state-of-the-art by providing the capability to collect, manage and use the slack time produced by task early completion. The slack time is then used as extra execution time by either high-criticality tasks to reduce mode-switches or low-criticality tasks to reduce task suspensions. Experimental results show that both
the number of mode-switches and the number of low-criticality task droppings are reduced by FMC-DS. In future work, we will consider slack generated from tasks’ early release and find more fine-grained mode-switch scheme of overrun handling to further reduce resource over-provisioning.

References


