Fixed-Priority Scheduling for Two-Phase Mixed-Criticality Systems

ZHENG LI, Western Illinois University
SHUIBING HE, Wuhan University

In this article, a two-phase execution model is proposed for mixed-criticality (MC) tasks. Different from traditional MC tasks with a computation phase only, the two-phase execution model requires a memory-access phase first to fetch the instructions and data, and then computation. Theoretical foundations are first established for a schedulability test under given memory-access and computation priority assignment. Based on the established theoretical conclusions, a two-stage priority assignment algorithm, which can find the best priority assignment for both memory-access and computation phases under fixed-priority scheduling, is further developed. Extensive experiments have been conducted and the experimental results validate the effectiveness of our proposed approach.

CCS Concepts: • **Computer systems organization → Real-time system specification**;

Additional Key Words and Phrases: Real-time, embedded system, mixed-criticality, memory-access

ACM Reference format:
Zheng Li and Shuibing He. 2017. Fixed-Priority Scheduling for Two-Phase Mixed-Criticality Systems. *ACM Trans. Embed. Comput. Syst.* 17, 2, Article 35 (November 2017), 20 pages.
https://doi.org/10.1145/3105921

1 INTRODUCTION

When designing complex embedded systems, in addition to the safety requirement, more and more non-functional requirements such as production cost, power consumption, and weight are enforced on the system design. Therefore, mixed-criticality (MC) design, which integrates tasks of different criticality levels on shared hardware platforms, is deemed to be the trend for future real-time and embedded systems (Burns and Davis 2013), especially in the automotive and avionics industry. Examples involve the unmanned aerial vehicle (UAV), which integrates the HI-criticality functionalities, such as flight-control tasks, and LO-criticality tasks, such as photo-capturing tasks, on the same platform (Barhorst et al. 2009).

MC design can achieve higher cost efficiency; the potential resource competition on the shared platform may cause the deployed tasks to miss their deadlines. HI-criticality tasks, such as flight-control tasks in the UAV system, are crucial in the entire system and a deadline miss will result in catastrophic consequences. To ensure system safety and guarantee HI-criticality tasks always meet their deadlines, two worst-case execution times, that is, worst-case execution time by design

Authors' addresses: Z. Li, School of Computer Sciences, Western Illinois University, 1 university cricle, Macomb, IL, USA; email: z-li2@wiu.edu; S. He, School of Computing, Wuhan University, LuoJiaShan Wuchang District, Wuhan, Hubei, China; email: heshuibing@whu.edu.cn.

https://doi.org/10.1145/3105921

and a more pessimistic one, worst-case execution time by certification, are set for a HI-criticality task (Baruah et al. 2011). Initially, an MC system is considered to be operated under the LO-mode; when any task executes over the designed worst-case execution time, the system changes to the HI-mode immediately and LO-criticality tasks may sacrifice their executions to guarantee HI-criticality task deadlines. Due to the safety and efficiency requirements, a MC system is schedulable if both of the following conditions are satisfied (Ekberg and Yi 2012): 1) both LO-criticality and HI-criticality tasks meet their deadlines under the LO-mode, and 2) HI-criticality tasks meet their deadlines under the HI-mode. Determining whether a given MC system is schedulable has been proven to be NP-hard (Baruah et al. 2011).

Extensive research has been conducted to address the MC schedulability issue and some well-known earliest deadline first (EDF)- and fixed-priority (FP)-based approaches have been published in the literature (Li et al. 2014; Baruah et al. 2011; Ekberg and Yi 2012, 2014; Su and Zhu 2013). To the best of our knowledge, the existing research work mainly focuses on computation intensive MC systems, that is, the tasks are assumed to be computation only.

Due to technology scaling, tens to hundreds of cores are being integrated on the same die to provide ever-increasing computing capacity (Villa et al. 2008). However, many-core chips are commonly equipped with shared physical memory for all the processing units and the potential memory-access competition may become the bottleneck. Though typical real-time tasks are computation intensive, their memory-access time to pre-fetch the instructions and required data cannot be ignored anymore on many-core platforms (Melani et al. 2015). Memory and processing units are isolated physical resources, and hence memory-access and computation phases are free of contention. Therefore, different tasks' memory-access and computation phases can be executed in parallel. Because of this property, the existing scheduling approaches cannot be directly applied to the MC tasks with both memory-access and computation phases.

With the above observations, our preliminary study on memory-aware MC scheduler was presented in Li and Wang (2016). We extend the topic in this article and address it more thoroughly. The major contributions made in this article are fourfold:

- —Propose a new two-phase execution model for MC tasks.
- —Establish new schedulability test theories to determine if a given two-phase MC task set is schedulable.
- —Develop a two-stage priority assignment strategy to assign tasks' memory-access and computation priorities under fixed-priority scheduling.
- —Set up experiments to validate the performance of our proposed approach under varied system configurations.

The rest of the article is organized as follows. Related work is summarized in Section 2. The system models and our target problem are defined in Section 3. The theoretical foundations for schedulability tests are established in Section 4, and our proposed two-stage priority assignment approach is presented in Section 5. In Section 6, we describe our experiment settings and discuss the experimental results. Finally, the conclusion and future research direction are pointed out in Section 7.

2 RELATED WORK

The research on MC scheduler started in recent years. In Baruah and Vestal (2008), Baruah discussed how to apply EDF algorithms in scheduling MC task sets. To ensure the schedulability of a MC task set, the EDF with virtual deadline (EDF-VD) scheduling algorithm, which assigns HI-criticality tasks reduced deadlines to ensure that HI-criticality tasks' deadline guarantee, was proposed in Baruah et al. (2012b). Later, Ekberg (Ekberg and Yi 2014) utilized demand-bound

function analysis and proposed a greedy approach to further improve the schedulability over EDF-VD algorithm. With Baruah's EDF-VD and Ekberg's greedy algorithm, LO-criticality task execution will be terminated when the system enters into the HI-mode, and hence the performance of LO-criticality tasks will be severely degraded. To provide some service guarantee of LO-criticality tasks under HI-mode, Liu (Liu et al. 2016) studied the imprecise MC model and proposed a sufficient test applied to this model. Gu (Zhao et al. 2015) further implemented preemption threshold in an EDF-VD algorithm for resource-constrained systems.

To improve the system's QoS when the system operates in HI-mode, Su (Su and Zhu 2013) proposed elastic task models (Buttazzo et al. 1998) and developed a strategy to increase LO-criticality task periods to reduce their competition against HI-criticality tasks. Lipari (Lipari and Buttazzo 2013) introduced a server-based approach that intellectually adjusted HI-criticality task deadlines to maximize the amount of capacity reclaimable by LO-criticality tasks. By noticing that postponing HI-criticality job execution can promote early execution of LO-criticality tasks, Park (Park and Kim 2011) developed a scheme called criticality based EDF (CBEDF), which delayed the execution of HI-criticality tasks as much as possible. In addition, Niz (de Niz et al. 2009) characterized the criticality inversion problem and presented a zero-slack scheduling scheme. They further combined the zero-slack schedule approach with the rate monotonic (RM) scheduling algorithm and developed the ZERO-SLACK-RM scheduling algorithm to maximize the execution of LO-criticality tasks. To further improve the quality-of-service of LO-criticality tasks, dynamic resource reservation-based approach was also studied in Li et al. (2014).

In addition to the EDF based scheduling algorithms, FP-based algorithms have also been studied extensively. Audsley's algorithm was proposed in Audsley (2001) as an optimal fixed-priority assignment, Vestal (Vestal 2007) later extended the algorithm to schedule multi-criticality tasks, and Baruah (Baruah and Chattopadhyay 2013) further applied Audsley's algorithm to schedule tasks in MC tasks with computation phase only.

However, all the above research focuses on computation-intensive tasks only. Since many-core technology is emerging, slow memory-access is becoming the bottleneck, and real-time community just started to look into this issue. Considering memory-access time is not ignorable, under the assumption that both memory-access and computation of the same task will be assigned at the same priority level, Melani et al. [2015] gave the response-time analysis for single criticality tasks with both memory-access and computation demand. In Melani et al. (2016), they further extended the discussion to improve the schedulability by assigning different priorities to each task's memory-access and computation phase. Different from the existing work presented in Melani et al. (2015, 2016) with focus on single criticality task set, that is, all tasks are of the same criticality, in this article, we are to address how to model and schedule MC tasks, that is, tasks are of different criticalites, with both memory-access and computation phases. Since the task models are different, the scheduling theories and algorithms developed for single criticality tasks cannot be directly applied to MC tasks. Therefore, new theories and algorithms have to be investigated for two-phase MC tasks.

3 MODELS AND PROBLEM FORMULATION

3.1 System Models

In this article, we focus on MC systems (Baruah et al. 2012a; de Niz et al. 2009) with tasks at two different criticality levels. In particular, for a given MC task set $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$, $\forall \tau_k : \tau_k = (\chi_k, E_k, M_k, C_k, T_k, D_k)$, where $\chi_k \in \{LO, HI\}$ indicates a task's criticality level. A MC system will run into either the LO-mode or the HI-mode execution. E_k is a pair $(E_k(LO), E_k(HI))$ where $E_k(\chi)$ is the worst-case execution time of τ_k under χ mode execution. For HI-criticality tasks,

$E_k(HI) \geq E_k(LO)$, while for LO-criticality tasks, $E_k(HI) = E_k(LO)$. The system starts at LO-mode, when any task τ_k executes up to $E_k(LO)$ time limit without signaling completion, the system will change to HI-mode immediately. After the switch point, HI-criticality tasks are still required even when they execute up to $E_k(HI)$. However, LO-criticality tasks are not required to meet any deadline under the HI-mode (Baruah et al. 2012a; de Niz et al. 2009; Ekberg and Yi 2014) and hence they will be suspended from further execution if competing for resources with HI-criticality tasks.

Different from the traditional MC tasks that are assumed to be computation only (Park and Kim 2011; Baruah et al. 2012b; Ekberg and Yi 2014; Su and Zhu 2013), we model each MC task as a two-phase execution: memory-access, which is to fetch the required instructions and data, and then computation. Though the model itself is simple, typical real-time tasks, such as image and signal processing tasks without data written back, fits this model well (Melani et al. 2015). Among which, $M_k = (M_k(LO), M_k(HI))$ and $M_k(\chi)$ indicates the worst-case memory access time under χ-mode. Similarly, C_k is a also pair $(C_k(LO), C_k(HI))$ and $C_k(\chi)$ is χ-mode worst-case computation time. E_k is the total execution time including memory-access and computation, that is, $E_k(\chi) = M_k(\chi) + C_k(\chi)$. A task τ_i generates potentially an infinite number of instances with at least T_k time units apart from each other. D_k is the relative deadline and tasks are assumed to have implicit deadlines. In addition, we assume a task's memory-access time remains the same under both the LO-mode and HI-mode execution, that is, $M_k(LO) = M_k(HI)$.

To satisfy the safety and efficiency requirements, a task set is defined to be MC schedulable if the following conditions are both satisfied (Baruah et al. 2012b):

(1) Both the HI-criticality and LO-criticality tasks meet their deadlines under LO-mode execution.
(2) HI-criticality tasks also meet their deadlines under HI-mode execution.

3.2 Problem Formulation

Before formulating the problem to be addressed, we first define the following notations to simplify the representation:
Q_M: task set memory access priority order.
$Q_M(\tau_i)$: the memory access priority of task τ_i in priority order Q_M.
Q_C: task set computation priority order.
$Q_C(\tau_i)$: the computation priority of task τ_i in priority order Q_C.
$hpm(\tau_i)$: the set of tasks with memory access priority higher than $Q_M(\tau_i)$.
$hpc(\tau_i)$: the set of tasks with computation priority higher than $Q_C(\tau_i)$.
$hpmc(\tau_i)$: the set of tasks with memory access priority higher than $Q_M(\tau_i)$ and computation priority higher than $Q_C(\tau_i)$.
$hpcH(\tau_i)$: the set of HI-criticality tasks with computation priority higher than $Q_C(\tau_i)$.
$R_M^L(\tau_i)$: worst-case response time of task τ_i in memory access phase under LO-mode.
$R_C^L(\tau_i)$: worst-case response time of task τ_i in computation phase under LO-mode.
$R^L(\tau_i)$: the worst-case response time of task τ_i under LO-mode including both memory access and computation phases.
$R^H(\tau_i)$: worst-case response time of task τ_i under HI-mode including both memory access and computation phases.
$u_L(\tau_i)$: task τ_i LO-mode utilization, where $u_L(\tau_i) = \frac{E_i(\text{LO})}{T_i}$.
$u_H(\tau_i)$: task τ_i HI-mode utilization, where $u_H(\tau_i) = \frac{E_i(\text{HO})}{T_i}$.
Γ_L: LO-criticality task subset that consists of all the LO-criticality tasks, that is, $\Gamma_L = \{\tau_k | \tau_k \in \Gamma \wedge \chi_k = LO\}$.

Γ_H: HI-criticality task subset that consists of all the HI-criticality tasks, that is, $\Gamma_H = \{\tau_k | \tau_k \in \Gamma \wedge \chi_k = HI\}$.
$U_\chi(\Gamma_L)$: LO-criticality task set χ-mode utilization, where $U_\chi(\Gamma_L) = \sum_{\tau_i \in \Gamma_L} u_\chi(\tau_i)$ and $\chi \in \{\text{HI, LO}\}$.
$U_\chi(\Gamma_H)$: HI-criticality task set χ-mode utilization, where $U_\chi(\Gamma_H) = \sum_{\tau_i \in \Gamma_H} u_\chi(\tau_i)$ and $\chi \in \{\text{HI, LO}\}$.

With the above notations, the problem we are to address in this article can be formulated as follows:

Problem 1. *Given a mixed-criticality task set $\Gamma = \{\Gamma_{LO}, \Gamma_{HI}\}$ with tasks that are of two-phase execution, that is, memory access first and then computation, develop a fixed-priority scheduler under which the mixed-criticality task set Γ is guaranteed to be schedulable.*

As mentioned above, a task set Γ is MC schedulable if (1) the worst case response time of both LO-criticality and HI-criticality tasks should not exceed their deadlines under LO-mode, and (2) the worst-case response time of HI-criticality tasks also should not exceed their deadlines under HI-mode. In other words, the following inequations must be satisfied:

$$\forall \tau_i \in \Gamma : R^L(\tau_i) \leq D_i \tag{1}$$

and

$$\forall \tau_i \in \Gamma_H : R^H(\tau_i) \leq D_i. \tag{2}$$

To design a fixed-priority scheduler for a two-phase MC task set, the key is to determine the memory-access priorities Q_M and computation priorities Q_C. In the following sections, we are to address the problem in two steps: (1) assuming Q_M and Q_C are given as *a priori*, develop schedulability test theories which determine whether a given MC task set is schedulable; (2) develop a priority strategy to determine the best Q_M and Q_C. It is worth pointing out that both the memory and computation phases are assumed to be preemptable (Li et al. 2012; Kaneko et al. 2003).

4 THEORETICAL FOUNDATION

Given Q_M and Q_C, Equations (1) and (2) can be used to determine if an MC task set is schedulable. In this section, before discussing how to calculate $R^L(\tau_i)$ and $R^H(\tau_i)$ in Equations (1) and (2), we give the following definitions first:

Definition 4.1 (Phase-Transition Instant). The time instant at which the task is transited from memory-access phase to computation phase.

Definition 4.2 (Mode-Transition Instant). The time instant at which the task is transited from LO-mode execution to HI-mode execution.

Definition 4.3 (Memory-Interfering Task Instance). Suppose J_k is a task instance of τ_k. If J_k's memory-access is interfered by some task instanceJ_i's memory-access, then J_i is said to be a memory-interfering task instance of J_k.

Definition 4.4 (Computation-Interfering Task Instance). Suppose J_k is a task instance of τ_k. If J_k's computation is interfered by some task instance J_i's computation, then J_i is said to be a computation-interfering task instance of J_k.

Definition 4.5 (Dual-Interfering Task Instance). Suppose J_k is a task instance released by τ_k. If some task instance J_i is both the memory-interfering task instance and computation-interfering task instance of J_k, then J_i is said to be a dual-interfering task instance of J_k.

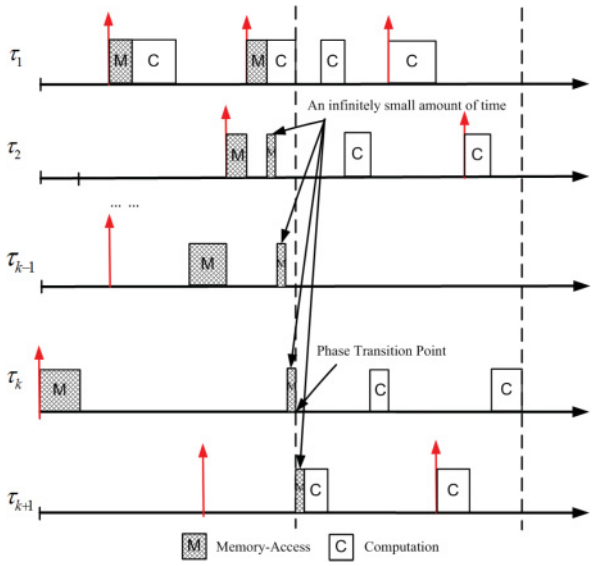

Fig. 1. Worst Case Response Time Scenario.

With the above definitions, now we discuss how to calculate a task's worst-case response time under LO-mode $R^L(\tau_k)$. Since a two-phase task's execution consists of both memory access and computation phases, $R^L(\tau_k)$ can be expressed as:

$$R^L(\tau_k) = R^L_M(\tau_k) + R^L_C(\tau_k), \tag{3}$$

where $R^L_M(\tau_k)$ and $R^L_C(\tau_k)$ are the worst-case response time of memory-access phase and computation phase, respectively.

For single criticality task set, that is, all the tasks are of the same criticality, Melani et al. (2016) proposed the critical instance theory to calculate tasks' worst-case execution time. For self-containment, we include the theory as follows:

Lemma 4.6. *Assuming tasks are of the same criticality, but each task's memory-access and computation phases may be at different priority levels, the worst-case response time of a task instance J_k released by task τ_k can be achieved when*

(1) Dual-interfering task instances complete their memory-access phases an infinitely small amount of time earlier than the phase-transition instant of J_k.
(2) Memory-interfering-only task instances complete their memory-access phases an infinitely small amount of time earlier than the phase-transition instance of J_k.
(3) Computation-interfering-only task instances complete their memory-access phases an infinitely small amount of time after the phase-transition instance of J_k.
(4) All computation-interfering task instances released after the phase-transition instance of J_k are with null memory-access phases.

Proof. The scenario having the conditions (1)–(4) is illustrated in Figure 1, where $Q_M = \{Q_M(\tau_1), Q_M(\tau_2), \ldots, Q_M(\tau_k), Q_M(\tau_{k+1})\}$, and $Q_C = \{Q_C(\tau_{k+1}), Q_C(\tau_1), Q_C(\tau_2), \ldots, Q_C(\tau_k)\}$, that is, tasks $\tau_1, \ldots, \tau_{k-1}$ have both higher memory-access and computation priority than τ_k, while task τ_{k+1} has higher computation but lower memory-access priority than τ_k. We prove that under this scenario both $R^L_M(\tau_k)$ and $R^L_C(\tau_k)$ of task instance J_k are maximized.

Conditions (1) and (2) indicate that all the tasks τ_i with higher memory-access priority, that is, $Q_M(\tau_i) > Q_M(\tau_k)$ complete their memory-access phases at the same time. To prove the $R_M^L(\tau_k)$ value is maximized under this scenario, we analogize the synchronous periodic release pattern which results in the maximum response time (Liu and Layland 1973), that is, shift right all higher memory-access phases until they complete an infinitely small amount of time ahead of J_k's phase-transition instant (Melani et al. 2015). Under such a scenario, all memory-interfering task instances' phase-transition instants are aligned with J_k's, which will result in the longest memory-access blocking time and hence $R_M^L(\tau_k)$ is maximized.

According to standard response time analysis (Joseph and Pandya 1986), $R_C^L(\tau_k)$ value is maximized when J_k and all the tasks τ_i with $Q_C(\tau_i) > Q_C(\tau_k)$ release their computation phases synchronously, which is stated as condition (3). In addition, condition (4) ensures the computation interfering task instances have null memory-access phases, which reveals that computation phases will be released as soon as possible and hence results in the longest blocking time of J_k's computation. All these conclude the proof. □

When an MC system executes under LO-mode, both LO-criticality and HI-criticality tasks are required to meet their deadlines; hence, we can treat all of these tasks at the same criticality level under LO-mode execution.

Based on these observations, similar with the Theorem 9 in Melani et al. (2016), task's worst case response time $R_M^L(\tau_k)$ and $R_C^L(\tau_k)$ can be derived using the following lemma.

Lemma 4.7. *Given an MC task set $\Gamma = \{\Gamma_L, \Gamma_H\}$, the memory access priority order Q_M and computation priority order Q_C, when a system executes under LO-mode, $\forall \tau_k \in \Gamma$, the worst-case response time of memory-access phase and computation phase can be calculated as*

$$R_M^L(\tau_k) = M_k(\mathrm{LO}) + \sum_{\tau_i \in hpm(\tau_k)} \left\lceil \frac{R_M^L(\tau_k)}{T_i} \right\rceil M_i(\mathrm{LO}) \tag{4}$$

and

$$R_C^L(\tau_k) = C_k(\mathrm{LO}) + \sum_{\tau_i \in hpc(\tau_k)} \left\lceil \frac{R_C^L(\tau_k) + R_M^L(\tau_i)}{T_i} \right\rceil C_i(\mathrm{LO}), \tag{5}$$

respectively.

Proof. With the standard response time analysis (Joseph and Pandya 1986), for a task instance J_k released by task τ_k, if its memory-access response time is $R_M^L(\tau_k)$, then the task with higher memory access priority, that is, $\tau_i \in hpm(\tau_k)$ will contribute $\lceil \frac{R_M^L(\tau_k)}{T_i} \rceil M_i(\mathrm{LO})$ blocking time. Hence, $R_M^L(\tau_k)$ can be calculated by fixed-point iteration of the following equation:

$$R_M^L(\tau_k) = M_k(\mathrm{LO}) + \sum_{\tau_i \in hpm(\tau_k)} \left\lceil \frac{R_M^L(\tau_k)}{T_i} \right\rceil M_i(\mathrm{LO}).$$

Next, we present how to calculate J_k's worst-case computation response. According to Lemma 4.6, $R_C^L(\tau_k)$ is maximized when J_k and all tasks $\tau_i \in hpc(\tau_k)$ release their computation phases synchronously. Under such a scenario, $\forall\, \tau_i \in hpc(\tau_k)$, the total number of interfering task instances to be completed will be

$$\left\lceil \frac{R_C^L(\tau_k) - (T_i - R_M^L(\tau_i))}{T_i} \right\rceil + 1,$$

where $R_M^L(\tau_i)$ is τ_i's memory-access response time and it is treated as the release jitter of τ_i's computation phase. As τ_i can be executed up to $C_i(\text{LO})$ under LO-mode, the total blocking time should be

$$\left\lceil \frac{R_C^L(\tau_k) + R_M^L(\tau_i)}{T_i} \right\rceil \cdot C_i(\text{LO}).$$

These conclude the proof of Lemma 4.7. □

As mentioned above, if any HI-criticality task executes over $E_i(\text{LO})$ time, then the system will be switched to HI-mode. Therefore, the system's mode change could be triggered by (1) a certain HI-criticality task τ_k itself or (2) some HI-criticality task other than τ_k. In the following, we present how to calculate a task's worst-case response time under HI-mode [i.e., $R_\alpha^H(\tau_k)$]. More specifically, Lemmas 4.8 and 4.9 are to calculate $R_\alpha^H(\tau_k)$ in the first case, and are demonstrated in detail through Lemmas 4.10–4.13 in the second case.

Lemma 4.8. *Given an MC task set $\Gamma = \{\Gamma_L, \Gamma_H\}$, the memory access priority order Q_M, and computation priority order Q_C, if the task τ_k triggers the system mode change from LO-mode to HI-mode, then its worst-case response time, that is, $R_\alpha^H(\tau_k)$ can be expressed as*

$$R_\alpha^H(\tau_k) = \max_{x \in \{E_k(\text{LO}), R^L(\tau_k)\}} R_\alpha^H(\tau_k, x), \tag{6}$$

where $R_\alpha^H(\tau_k, x)$ represents the worst-case response time under HI-mode when the mode-transition instant happens at time x and it can be calculated by fixed-point iteration of the following equation:

$$\begin{aligned} R_\alpha^H(\tau_k, x) &= C_k(\text{HI}) - C_k(\text{LO}) + x \\ &\quad + \sum_{\tau_i \in hpcH(\tau_k)} \left\lceil \frac{R_\alpha^H(\tau_k, x) - x + R_M^L(\tau_i)}{T_i} \right\rceil C_i(\text{HI}), \end{aligned} \tag{7}$$

among which, $hpcH(\tau_k)$ indicates the set of HI-criticality tasks with computation priority higher than τ_k.

Proof. When a HI-criticality task τ_k runs over $E_k(LO)$, the system will change to HI-mode. Since τ_k's worst-case response time under LO-mode is $R^L(\tau_k)$, the system mode could occur at any time x between $E_k(LO)$ and $R^L(\tau_k)$, that is, $E_k(LO) \le x \le R^L(\tau_k)$. With standard response time analysis, task τ_k's worst-case response time under HI-mode when system mode changes at x can be obtained as

$$\begin{aligned} R_\alpha^H(\tau_k, x) &= E_k(\text{LO}) + B_k^L(x) + C_k(\text{HI}) - C_k(\text{LO}) \\ &\quad + B_k^H(R_\alpha^H(\tau_k, x) - x), \end{aligned} \tag{8}$$

where $B_k^L(x)$ denotes the blocking time from tasks with higher priority before τ_k's mode transition instant, $C_k(HI) - C_k(LO)$ is task overrun under the HI-mode, and $B_k^H(R_\alpha^H(\tau_k, x) - x)$ is the blocking time from tasks with higher computation priority. Suppose system changes mode at time x, then we have

$$x = E_k(\text{LO}) + B_k^L(x). \tag{9}$$

Since $E_k(LO) \ge M_k(HI)$, which indicates that the task has completed its memory-access phase under the LO-mode, the whole HI-mode execution is to finish the computation overrun. Task τ_k's blocking time under HI-mode is contributed by HI-criticality tasks with higher computation priority. Similar to the proof given in Lemma 4.6, the longest blocking time will be achieved if the following conditions are satisfied:

(1) $\forall \tau_i \in hpcH(\tau_k)$ releases the computation phase at τ_k's mode transition instant x.
(2) $\forall \tau_i \in hpcH(\tau_k)$ has null memory access phase after τ_k's mode transition instant x.
(3) $\forall \tau_i \in hpcH(\tau_k)$'s computation executes up to $C_i(\text{HI})$ time units.

With condition (1), by analogy with the response-time analysis for classic sporadic tasks with release jitter (Melani et al. 2015), $R_M^L(\tau_i)$ can be treated as the release jitter and we have

$$B_k^H(R_\alpha^H(\tau_k,x)-x) = \sum_{\tau_i \in hpcH(\tau_k)} \left(\left\lceil \frac{R_\alpha^H(\tau_k,x)-x-(T_i-R_M^L(\tau_i))}{T_i} \right\rceil + 1 \right) C_i(\text{HI}). \tag{10}$$

With Equations (9) and (10), Equation (8) can be rewritten as

$$R_\alpha^H(\tau_k,x) = C_k(\text{HI}) - C_k(\text{LO}) + x + \sum_{\tau_i \in hpcH(\tau_k)} \left\lceil \frac{R_\alpha^H(\tau_k,x)-x+R_M^L(\tau_i)}{T_i} \right\rceil C_i(\text{HI}). \tag{11}$$

Since x could be any time between $E_k(\text{LO})$ and $R^L(\tau_k)$, the task's worst-case response time under HI-mode can be calculated as

$$R_\alpha^H(\tau_k) = \max_{x \in \{E_k(\text{LO}), R^L(\tau_k)\}} R_\alpha^H(\tau_k,x).$$

These conclude the proof of Lemma 4.8. □

With Lemma 4.8, $R_\alpha^H(\tau_k)$ can be obtained to determine if the task is schedulable under the HI-mode. However, to obtain $R_\alpha^H(\tau_k)$, fixed-point iteration of Equation (7) for every possible $x \in \{E_k(\text{LO}), R^L(\tau_k)\}$ must be calculated, which is time consuming. Next, we present Lemma 4.9 to simplify the above calculation.

Lemma 4.9. *Given an MC task set $\Gamma = \{\Gamma_L, \Gamma_H\}$, the memory access priority order Q_M and computation priority order Q_C, if the task τ_k triggers the system mode change from LO-mode to HI-mode, then its worst-case response time under HI-mode, that is, $R_\alpha^H(\tau_k)$ can be expressed as:*

$$R_\alpha^H(\tau_k) = \lambda + R^L(\tau_k), \tag{12}$$

where λ is calculated by fixed-point iteration of the following equation:

$$\lambda = C_k(\text{HI}) - C_k(\text{LO}) + \sum_{\tau_i \in hpcH(\tau_k)} \left\lceil \frac{\lambda + R_M^L(\tau_i)}{T_i} \right\rceil C_i(\text{HI}). \tag{13}$$

Proof. We prove this Lemma by simplifying Equation (6) to Equation (12). By setting

$$\lambda = R_\alpha^H(\tau_k,x) - x,$$

then Equation (11) can be rewritten as Equation (13).

By applying fixed-point iteration of Equation (13), we can obtain the value of λ, which is independent of the value of x. Since $R_\alpha^H(\tau_k,x) = \lambda + x$ and $E_k(\text{LO}) \le x \le R^L(\tau_k)$, we have

$$\max_{x \in \{E_k(\text{LO}), R^L(\tau_k)\}} R_\alpha^H(\tau_k,x) = \lambda + R^L(\tau_k).$$

According to Equation (6), we can further get:

$$R_\alpha^H(\tau_k) = \lambda + R^L(\tau_k).$$

These conclude the proof. □

It is not hard to find that, with Lemma 4.9, $R_{\alpha}^{H}(\tau_k)$ can be obtained by applying fixed-point iteration of Equation (13) only once. Comparing with Lemma 4.8, the time cost is greatly reduced.

As stated above, Lemmas 4.8 and 4.9 are only applied for such τ_k, which triggers the system mode change. If some other HI-criticality task but not τ_k initiates the system mode switch, then at the mode transition instant, τ_k could be within (a) memory-access phase or (b) computation phase.

Lemma 4.10. *Given an MC task set $\Gamma = \{\Gamma_L, \Gamma_H\}$, the memory access priority order Q_M and computation priority order Q_C, if the task τ_k is within memory-access phase at the mode transition instant, then its worst-case response time under HI-mode, that is, $R_{\beta}^{H}(\tau_k)$ can be expressed as*

$$R_{\beta}^{H}(\tau_k) = R_{M}^{L}(\tau_k) + R_{C}^{H}(\tau_k), \tag{14}$$

where $R_{C}^{H}(\tau_k)$ can be calculated by fixed-point iteration of the following equation:

$$R_{C}^{H}(\tau_k) = C_k(\mathrm{HI}) + \sum_{\tau_i \in hpcH(\tau_k)} \left\lceil \frac{R_{C}^{H}(\tau_k) + R_{M}^{L}(\tau_i)}{T_i} \right\rceil C_i(\mathrm{HI}). \tag{15}$$

Proof. After the mode transition instant x, all the LO-criticality tasks are suspended from further execution. If the task τ_k is still in memory-access phase at time x, suppose the memory-access phase response time of the task τ_k under HI-mode is $R_{M}^{H}(\tau_k)$, then we should have $R_{M}^{H}(\tau_k) \leq R_{M}^{L}(\tau_k)$. This is because $M_L(\tau_k) = M_H(\tau_k)$, but less memory-access interference under HI-mode due to the suspension of LO-criticality tasks.

In addition, the computation phase will start after the memory-access and the worst-case response time of τ_k's computation phase should be $R_{C}^{H}(\tau_k)$:

$$\begin{aligned} R_{C}^{H}(\tau_k) &= C_k(\mathrm{HI}) + \sum_{\tau_i \in hpcH(\tau_k)} \left(\left\lceil \frac{R_{C}^{H}(\tau_k) - (T_i - R_{M}^{H}(\tau_k))}{T_i} \right\rceil + 1 \right) C_i(\mathrm{HI}) \\ &= C_k(\mathrm{HI}) + \sum_{\tau_i \in hpcH(\tau_k)} \left\lceil \frac{R_{C}^{H}(\tau_k) + R_{M}^{H}(\tau_i)}{T_i} \right\rceil C_i(\mathrm{HI}), \end{aligned}$$

where $R_{M}^{H}(\tau_k)$ is treated as the release time of the computation phase.

Since the total response time of task τ_k under HI-mode ($R_{\beta}^{H}(\tau_k)$) includes memory-access and computation, it should be $R_{\beta}^{H}(\tau_k) = R_{M}^{H}(\tau_k) + R_{C}^{H}(\tau_k)$. As $R_{M}^{H}(\tau_k) \leq R_{M}^{L}(\tau_k)$, $R_{\beta}^{H}(\tau_k)$ achieves the maximum value at $R_{M}^{H}(\tau_k) = R_{M}^{L}(\tau_k)$. □

Lemma 4.11. *Given an MC task set $\Gamma = \{\Gamma_L, \Gamma_H\}$, the memory access priority order Q_M and computation priority order Q_C, if the task τ_k has finished memory-access phase but not yet started the computation phase at the mode transition instant x, then its worst-case response time under HI-mode, that is, $R_{\gamma}^{H}(\tau_k)$ can be calculated as:*

$$R_{\gamma}^{H}(\tau_k) = \max_{x \in [M_k(\mathrm{LO}), R^L(\tau_k) - C_k(\mathrm{LO})]} (R_{\gamma}^{H}(\tau_k, x)), \tag{16}$$

where $R_{\gamma}^{H}(\tau_k, x)$ represents the worst-case response time under HI-mode when the mode-transition instant happens at time x, and it can be calculated by fixed-point iteration of the following equation:

$$R_{\gamma}^{H}(\tau_k, x) = x + C_k(\mathrm{HI}) + \sum_{\tau_i \in hpcH(\tau_k)} \left\lceil \frac{R_{\gamma}^{H}(\tau_k, x) - x + R_{M}^{L}(\tau_i)}{T_i} \right\rceil C_i(\mathrm{HI}). \tag{17}$$

Proof. We follow the similar proof as used for lemma 4.8.

If task τ_k has finished memory-access phase but not yet starts the computation at mode transition instant x, then we have $M_k(\text{LO}) \le x \le R^L(\tau_k) - C_k(\text{LO})$.

Since task τ_k's computation phase is not started, $R^H_\gamma(\tau_k, x)$ should be at least $x + C_k(\text{HI})$. In addition, within the HI-mode duration $R^H_\gamma(\tau_k, x) - x$, the inference from higher computation phases can be expressed as :

$$B^H_k(R^H_\gamma(\tau_k, x) - x) = \sum_{\tau_i \in hpcH(\tau_k)} \left(\left\lceil \frac{R^H_\gamma(\tau_k, x) - x - (T_i - R^L_M(\tau_i))}{T_i} \right\rceil + 1 \right) C_i(\text{HI})$$
$$= \sum_{\tau_i \in hpcH(\tau_k)} \left\lceil \frac{R^H_\gamma(\tau_k, x) - x + R^L_M(\tau_i)}{T_i} \right\rceil C_i(\text{HI})$$

Based on the above analysis, $R^H_\gamma(\tau_k, x)$ can be calculated as:

$$R^H_\gamma(\tau_k, x) = x + C_k(\text{HI}) + B^H_k(R^H_\gamma(\tau_k, x) - x).$$

All the above concludes the proof. □

Lemma 4.12 is proposed to simplify the calculation of Lemma 4.11.

Lemma 4.12. *Given an MC task set $\Gamma = \{\Gamma_L, \Gamma_H\}$, the memory access priority order Q_M and computation priority order Q_C, if the task τ_k has finished memory-access phase but not yet started the computation phase at the mode transition instant x, then its worst-case response time under HI-mode, that is, $R^H_\gamma(\tau_k)$ can be calculated as:*

$$R^H_\gamma(\tau_k) = \lambda + R^L(\tau_k) - C_k(\text{LO}), \tag{18}$$

where λ is calculated by fixed-point iteration of the following equation:

$$\lambda = C_k(\text{HI}) + \sum_{\tau_i \in hpcH(\tau_k)} \left\lceil \frac{\lambda + R^L_M(\tau_i)}{T_i} \right\rceil C_i(\text{HI}). \tag{19}$$

Proof. Following the analogous proof given in Lemma 4.9, Lemma 4.12 can be proved by setting $\lambda = R^H_\gamma(\tau_k, x) - x$. As $x \le R^L(\tau_k) - C_k(\text{LO})$, the maximum of $R^H_\gamma(\tau_k)$ will be achieved when $x = R^L(\tau_k) - C_k(\text{LO})$. □

Lemma 4.13. *Given an MC task set $\Gamma = \{\Gamma_L, \Gamma_H\}$, the memory access priority order Q_M and computation priority order Q_C, if the task τ_k is within computation phase at the mode transition instant, then its worst-case response time under HI-mode, that is, $R^H_\delta(\tau_k)$ can be calculated as*

$$R^H_\delta(\tau_k) = C_k(\text{HI}) + R^L(\tau_k) - C_k(\text{LO})$$
$$+ \sum_{\tau_i \in hpcH(\tau_k)} \left\lceil \frac{R^H_\delta(\tau_k) - (R^L(\tau_k) - C_k(\text{LO})) + R^L_M(\tau_i)}{T_i} \right\rceil C_i(\text{HI}). \tag{20}$$

Proof. When the task τ_k is within computation phase at the mode transition instant x, theoretically, we could have $M_k(\text{LO}) < x \le R^L(\tau_k)$. However, under the worst-case scenario, the execution of computation phase could be delayed until $x = R^L(\tau_k) - C_k(\text{LO})$. Therefore, we only focus on $R^L(\tau_k) - C_k(\text{LO}) \le x \le R^L(\tau_k)$ and hence we have

$$R^H_\delta(\tau_k) = \max_{x \in \{R^L(\tau_k) - C_k(\text{LO}), R^L(\tau_k)\}} R^H_\delta(\tau_k, x). \tag{21}$$

At the mode transition instant x, there must be at most $R^L(\tau_k) - x$ computation supposed to be completed under LO-mode but not yet finished. Hence, after system changes to HI-mode, the

computation phase to be finished is at most $\nabla = C_k(\text{HI}) - C_k(\text{LO}) + R^L(\tau_k) - x$. Therefore, the worst-case response time of task τ_k can be calculated as

$$R^H_\delta(\tau_k, x) = \nabla + x + \sum_{\tau_i \in hpcH(\tau_k)} \left\lceil \frac{R^H_\delta(\tau_k, x) - x + R^L_M(\tau_i)}{T_i} \right\rceil C_i(\text{HI})$$

$$= C_k(\text{HI}) - C_k(\text{LO}) + R^L(\tau_k) + \sum_{\tau_i \in hpcH(\tau_k)} \left\lceil \frac{R^H_\delta(\tau_k, x) - x + R^L_M(\tau_i)}{T_i} \right\rceil C_i(\text{HI}). \quad (22)$$

Since $R^L(\tau_k) - C_k(\text{LO}) \le x \le R^L(\tau_k)$, $R^H_\delta(\tau_k, x)$ achieves the maximum value when $x = R^L(\tau_k) - C_k(\text{LO})$.

All the above conclude the proof. □

To consolidate the above-mentioned cases together and give a uniform formula to calculate the HI-mode worst case response time of the task τ_k, we have

$$R^H(\tau_k) = \max\{R^H_\alpha(\tau_k), R^H_\beta(\tau_k), R^H_\gamma(\tau_k), R^H_\delta(\tau_k)\}$$

Lemma 4.14. *Given an MC task set $\Gamma = \{\Gamma_L, \Gamma_H\}$, the memory access priority order Q_M and computation priority order Q_C, $\forall \tau_k \in \Gamma_H$, its worst-case response time under HI-mode, that is, $R^H(\tau_k)$ can be obtained by fixed-point iteration of the following equation:*

$$R^H(\tau_k) = C_k(\text{HI}) + R^L(\tau_k) - C_k(\text{LO}) + \sum_{\tau_i \in hpcH(\tau_k)} \left\lceil \frac{R^H(\tau_k) - (R^L(\tau_k) - C_k(\text{LO})) + R^L_M(\tau_i)}{T_i} \right\rceil C_i(\text{HI}). \quad (23)$$

Proof. Since the Equations (23) and (20) are the same, in the following we prove that $R^H(\tau_k) = \max\{R^H_\alpha(\tau_k), R^H_\beta(\tau_k), R^H_\gamma(\tau_k), R^H_\delta(\tau_k)\} = R^H_\delta(\tau_k)$.

1) $R^H_\alpha(\tau_k) \le R^H_\delta(\tau_k)$: According to Equation (12), $R^H_\alpha(\tau_k) = \lambda + R^L(\tau_k)$ where λ is calculated by Equation (13). Plugging $\lambda = R^H_\alpha(\tau_k) - R^L(\tau_k)$ in Equation (13), we have

$$R^H_\alpha(\tau_k) = C_k(\text{HI}) + R^L(\tau_k) - C_k(\text{LO}) + \sum_{\tau_i \in hpcH(\tau_k)} \left\lceil \frac{R^H_\alpha(\tau_k) - R^L(\tau_k) + R^L_M(\tau_i)}{T_i} \right\rceil C_i(\text{HI})$$

$$\le C_k(\text{HI}) + R^L(\tau_k) - C_k(\text{LO}) + \sum_{\tau_i \in hpcH(\tau_k)} \left\lceil \frac{R^H_\alpha(\tau_k) - (R^L(\tau_k) - C_k(\text{LO})) + R^L_M(\tau_i)}{T_i} \right\rceil C_i(\text{HI}).$$

Since $R^H_\delta(\tau_k)$ is calculated using Equation (20), it is not hard to find that $R^H_\alpha(\tau_k) \le R^H_\delta(\tau_k)$.

2) $R^H_\beta(\tau_k) \le R^H_\gamma(\tau_k)$: Comparing with Equations (15) and (19), the $R^H_C(\tau_k)$ obtained from Equation (15) and λ calculated from Equation (19) should have the same value. Since $R^L(\tau_k) - C_k(\text{LO}) \ge R^L_M(\tau_k)$, according to Equations (14) and (18), we have $R^H_\beta(\tau_k) \le R^H_\gamma(\tau_k)$.

3) $R^H_\gamma(\tau_k) = R^H_\delta(\tau_k)$: Based on Equation (18), by plugging $\lambda = R^H_\gamma(\tau_k) - (R^L(\tau_k) - C_k(\text{LO}))$ into Equation (19), we can get

$$R^H_\gamma(\tau_k) = C_k(\text{HI}) + (R^L(\tau_k) - C_k(\text{LO})) + \sum_{\tau_i \in hpcH(\tau_k)} \left\lceil \frac{R^H_\gamma(\tau_k) - (R^L(\tau_k) - C_k(\text{LO})) + R^L_M(\tau_i)}{T_i} \right\rceil C_i(\text{HI}),$$

which is the same formula used to calculate $R^H_\delta(\tau_k)$, hence $R^H_\gamma(\tau_k) = R^H_\delta(\tau_k)$.

Table 1. Task Parameters

τ_i	χ_i	M_i	$C_i(LO)$	$C_i(HI)$	$E_i(LO)$	$E_i(HI)$	T_i
τ_1	HI	2	1	2	3	4	5
τ_2	HI	1	4	7	5	8	10
τ_3	LO	1	5	5	6	6	20

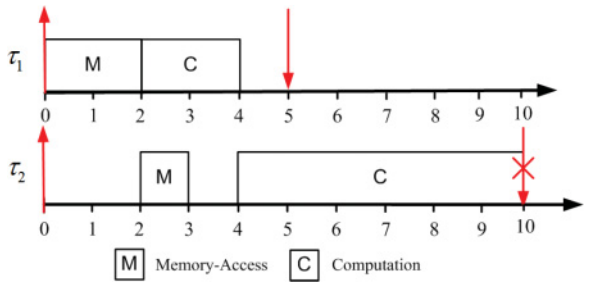

Fig. 2. Task schedule with $Q_M(\tau_1) > Q_M(\tau_2)$ and $Q_C(\tau_1) > Q_C(\tau_2)$.

With the above analysis, we have $R^H(\tau_k) = \max\{R^H_\alpha(\tau_k), R^H_\beta(\tau_k), R^H_\gamma(\tau_k), R^H_\delta(\tau_k)\} = R^H_\delta(\tau_k, x)$. □

5 MEMORY-PROCESSOR PRIORITY CO-ASSIGNMENT FOR MIXED-CRITICALITY TASK SET

We have established the theories to decide whether an MC task set is schedulable, assuming Q_M and Q_C are known *a priori.* In this section, we discuss how to determine Q_M and Q_C.

Example 5.1. Consider an MC task set $\Gamma = \{\Gamma_L, \Gamma_H\}$, where $\Gamma_H = \{\tau_1, \tau_2\}$, $\Gamma_L = \{\tau_3\}$, and the parameters are listed in Table 1.

For traditional MC tasks with computation only, adaptive MC (AMC) scheduling algorithm (Baruah et al. 2011), which assigns each task a single fixed priority, has been proved to be the optimal one regarding to the schedulability. However, for MC tasks having both memory-access and computation phases, any single priority assignment, that is, each task has the memory-access and computation phases assigned at the same priority level, will not be the optimal one and a counter-example is given as follows:

Considering the following scenario: τ_1 runs two time units for memory access and two time units for computation, task τ_2 executes one time unit for memory access and *seven* time units for computation. It is not hard to find that the system will operate under HI-mode and task τ_3 will be suspended. Hence, we focus on the scheduling of HI-criticality tasks τ_1 and τ_2 only.

By using single priority assignment, that is, a task's memory-access and computation phases will be assigned at the same priority level, there will be only two possible options: 1) $Q_M(\tau_1) > Q_M(\tau_2)$ and $Q_C(\tau_1) > Q_C(\tau_2)$, and 2) $Q_M(\tau_1) < Q_M(\tau_2)$ and $Q_C(\tau_1) < Q_C(\tau_2)$. The task set scheduling order under options 1 and 2 are depicted in Figure 2 and Figure 3, respectively. From which, it is not hard to find that the MC task set is unschedulable under either option 1 or 2. Therefore, we can conclude that this task set is unschedulable under any single priority assignment.

However, this MC task set is actually schedulable if the memory-access and computation phases are assigned at different priority levels. As illustrated in Figure 4 , both tasks meet their deadlines if $Q_M(\tau_1) > Q_M(\tau_2)$ and $Q_C(\tau_1) < Q_C(\tau_2)$.

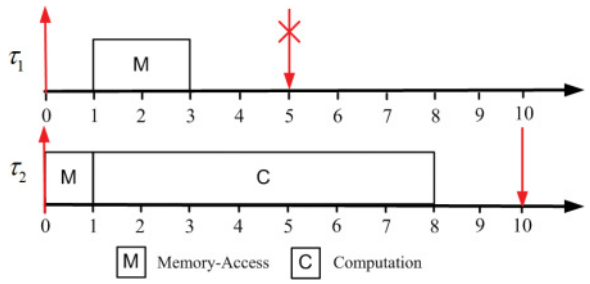

Fig. 3. Task schedule with $Q_M(\tau_1) < Q_M(\tau_2)$ and $Q_C(\tau_1) < Q_C(\tau_2)$.

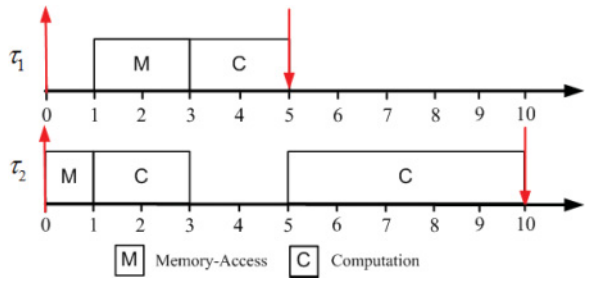

Fig. 4. Task schedule with $Q_M(\tau_1) < Q_M(\tau_2)$ and $Q_C(\tau_1) > Q_C(\tau_2)$.

Inspired by Example 5.1, instead of single priority assignment, in the following we present our two-stage priority assignment algorithm, which may assign task's memory-access and computation at different priority levels to improve the schedulability performance.

According to Equation (5), to calculate the task τ_k's computation response time $R_C^L(\tau_k)$, the memory access response time of all such task $\tau_i \in hpc(\tau_k)$ must be known *a priori*. Therefore, in our proposed approach, we first assign the memory-access priorities and then computation priorities.

Two new terms are defined to simplify the following representation:

(1) $D_M(\tau_i)$: memory-access deadline, where $D_M(\tau_i) = D_i - C_i(HI)$.
(2) $S_M(\tau_i)$: memory-access slack time, where $S_M(\tau_i) = D_M(\tau_i) - R_M^L(\tau_i)$.

For a task τ_i, its memory access phase must be finished at least $C_i(HI)$ time ahead of its deadline; otherwise, it is impossible to finish the computation phase before the deadline. Hence, the deadline of its memory-access phase is set as $D_M(\tau_i) = D_i - C_i(HI)$. In addition, $S_M(\tau_i)$ indicates the tightness of τ_i's memory-access deadline and a larger $S_M(\tau_i)$ implies the task τ_i's computation phase is more likely to be completed before its deadline. Based on the above analysis, our memory priority assignment (MPA) approach can be highlighted as follows:

(1) Memory-access priorities are assigned from lowest to highest order; if the current lowest priority is assigned, the next higher one becomes the lowest available one.
(2) Assign the lowest available memory-access priority to the task with $R_M^L(\tau_i) \leq D_M(\tau_i)$. Under such assignment, ties are broken by giving priority to the one with largest $S_M(\tau_i)$.
(3) Repeat the above steps until all tasks' memory-access phases are assigned.

With MPA algorithm, the memory-access priorities Q_M can be determined, and hence each task τ_i's memory-access response time $R_M^L(\tau_i)$ can be calculated using Equation (4).

The next step is to assign the computation priorities Q_C. If a task's computation phase can meet deadline under both LO-mode and HI-mode, this task is schedulable. With the above observation, our proposed computation priority assignment (CPA) algorithm can be summarized in the following steps:

(1) Computation priorities are assigned from lowest to highest order; if the current lowest priority is assigned, the next higher one becomes the lowest available one.
(2) Assign the lowest available computation priority to a task that has the LO-mode and HI-mode response time, that is, $R^L(\tau_i)$ and $R^H(\tau_i)$ satisfy Equations (1) and (2), respectively; ties are broken arbitrarily.
(3) Repeat the above steps until all tasks' computation phases are assigned.

With the above analysis, the details of our proposed memory-access and computation priority assignment (MCPA) algorithm are illustrated in Algorithm 1. Among which, lines 2–11 are for the MPA and 12–19 are for the CPA. The algorithm will return failure (line 9 and line 17) if either MPA or CPA is not able to find a valid priority assignment.

ALGORITHM 1: MCPA($\Gamma = \{\Gamma_L, \Gamma_H\}$)

```
1  set Q_C = ∅, Q_M = ∅;
2  for mp := 0; mp < |Γ|; mp++ do
3      find the task subset Ω_M where τ_i ∈ Ω_M with R^L_M(τ_i) ≤ D_M(τ_i) if Q_M(τ_i) = mp;
4      if Ω_M ≠ ∅ then
5          find the task τ_k ∈ Ω_M with the largest S_M(τ_k);
6          set Q_M(τ_k) = mp;
7      end
8      else
9          return FAILURE;
10     end
11 end
12 for cp := 0; cp < |Γ|; cp++ do
13     if ∃τ_k ∈ Γ_L: R^L(τ_k) ≤ D_k or ∃τ_k ∈ Γ_H: R^L(τ_k) ≤ D_k and R^H(τ_k) ≤ D_k with Q_C(τ_k) = cp
       then
14         set Q_C(τ_k) = cp;
15     end
16     else
17         return FAILURE;
18     end
19 end
20 return Q = {Q_M, Q_C}.
```

6 EVALUATION

In this section, we conduct a set of experiments to evaluate the schedulability performance of proposed MCPA algorithm and the following approaches are set as the baselines:

(1) 2BF: two-stage brute-force search, that is, brute-force searching the best among all the possible memory-access and computation priority assignments.

(2) HEUR-DP-LO: heuristic priority assignment proposed in Melani et al. (2016) by setting $C_k(\text{LO})$ as the computation time for priority assignment.
(3) HEUR-DP-HI: similar with HEUR-dp-LO, but using $C_k(\text{HI})$ as the computation time for priority assignment.

In addition, AMC (Baruah et al. 2011) algorithm is also added in the comparisons. As AMC only can schedule single-phase MC tasks, we execute memory-access and computation phases sequentially when applying AMC.

It is worth pointing out that 2BF algorithm always returns the optimal solution, but it is time unafforable. If there are N tasks, 2BF has to traverse all the possible N! × N! different assignment options. However, the search space of AMC and our proposed MCPA algorithm will only be $O(N^2)$, which is much smaller than that of 2BF.

6.1 Experimental Setting

In the following experiments, HI- and LO-criticality tasks are generated using UUniFast algorithm (Bini and Buttazzo 2005), which can create an unbiased task set in the sense that the utilizations of the tasks are uniformly distributed. In particular, the following steps are used to generate a valid task set:

— The utilization of a HI-criticality and LO-criticality task set are $U_H(\Gamma_H)$ and $U_L(\Gamma_L)$, respectively. The individual task utilizations $u_H(\tau_i)$ and $u_L(\tau_i)$ are uniformly distributed in $[0, U_H(\Gamma_H)]$ and $[0, U_L(\Gamma_L)]$, respectively;
— Task's period T_i is randomly selected from $[50, 200]$;
— HO-criticality task's execution time $E_i(HI)$ is set as $T_i \cdot u_H(\tau_i)$ and $E_i(LO) = \lambda \cdot E_i(HI)$, where λ is a random value within the range $[0.4, 0.8]$;
— LO-criticality task's execution time $E_i(HI)$ is set as $T_i \cdot u_L(\tau_i)$ and $E_i(LO) = E_i(HI)$;
— Task's memory-access time $M_i(HI) = M_i(LO) = \gamma \cdot E_i(LO)$, where $\gamma \leq 1$ is called the memory access ratio;
— Task's computation time $C_i(\chi) = E_i(\chi) - M_i(\chi)$.

We define a metric called schedulability ratio, that is, the number of task sets passing the schedulability test over the total number of generated task sets, to quantify the performance of the compared algorithms. In addition, the comparisons will be made through the following different aspects:

(1) Impact of HI-mode utilization $U_H(\Gamma_H)$
(2) Impact of LO-mode utilization $U_L(\Gamma_L)$
(3) Sensitivity to memory access ratio r
(4) Sensitivity to the task set size $|\Gamma|$

6.2 Experiment Results and Discussions

In the following experiments, except the last one, six tasks are generated in each task set. Among which, three are of HI-criticality and the other three are of LO-criticality. The impact of task set size will be evaluated as the last set of experiments. The results shown in the following figures are the average values of repeating the experiments with 100 different task sets.

6.2.1 Impact of HI-Mode Utilization $U_H(\Gamma_H)$. In the first set of experiments, we set the memory access ratio $r = 0.6$, LO-criticality task set utilization $U_L(\Gamma_L) = 0.5$, and change HI-criticality task set utilization $U_H(\Gamma_H)$ from 0.4 to 1.0.

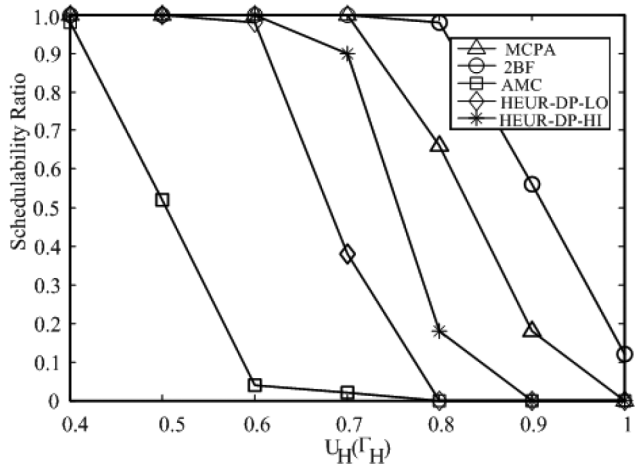

Fig. 5. Schedulability ratio comparison under varied $U_H(H)$ with $U_L(L) = 0.5$.

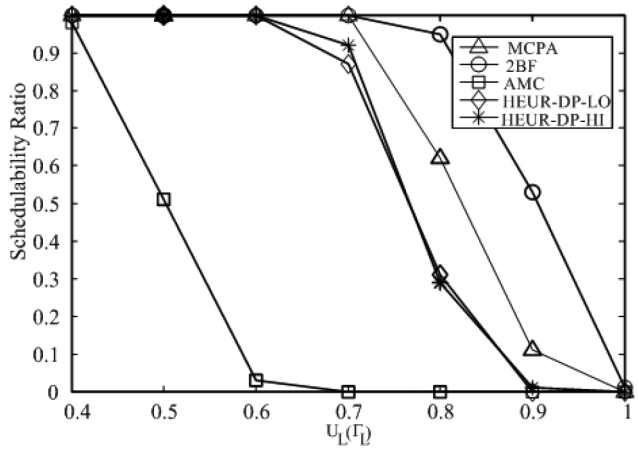

Fig. 6. Schedulability ratio comparison under varied $U_L(L)$ with $U_H(H) = 0.5$.

The results depicted in Figure 5 reveal that higher $U_H(\Gamma_H)$ results in lower schedulability ratio for all compared algorithms. Among them, AMC has the worst schedulability performance—this is because AMC assumes memory-access and computation phases to be executed sequentially. Our proposed MCPA algorithms always perform better than HEUR-DP-LO and HEUR-DP-HI under varied $U_H(\Gamma_H)$. When $U_H(\Gamma_H) = 0.8$, the schedulability ratio of MCPA is over 50% higher than that of HEUR-DP-HI. These are due to the fact that MC tasks have two execution times, that is, LO-mode and HI-mode execution times, but only one is counted in the priority assignment of HEUR-DP-HI/HEUR-DP-LO algorithm. 2BF performs the best, which can achieve up to 15% higher schedulability ratio than MCPA, but it has to exhaustively search all the possible options, which is time unaffordable.

6.2.2 Impact of LO-Mode Utilization $U_L(\Gamma_L)$. In this set of experiments, we set $r = 0.6, U_H(\Gamma_H) = 0.5$, and vary the LO-mode utilization $U_L(\Gamma_L)$ from 0.4 to 1.0 to evaluate the impact of $U_L(\Gamma_L)$.

The experiment results are shown in Figure 6. Analogous to the trend shown in Figure 5, 2BF is the best, as it can find the optimal solution. Our proposed MCPA algorithm has better

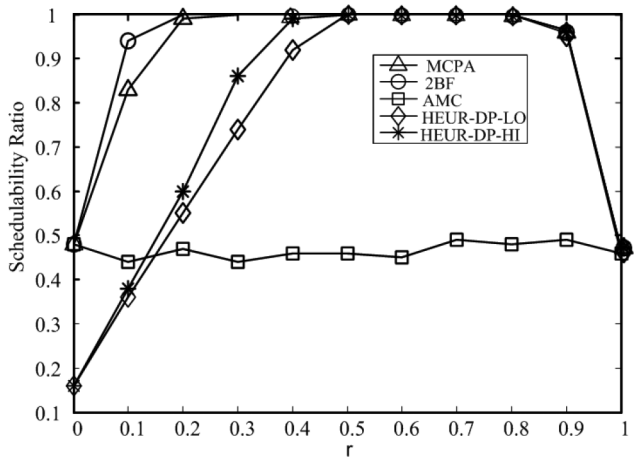

Fig. 7. Schedulability ratio comparison under varied memory-access ratio.

schedulability performance than both HEUR-DP-LO and HEUR-DP-HI algorithms, especially under higher $U_L(\Gamma_L)$. When $U_L(\Gamma_L) = 0.8$, schedulability ratio of MCPA algorithm is over 30% higher than that of HEUR-DP-HI. AMC algorithm is almost unable to schedule any task set when $U_L(\Gamma_L) > 0.7$.

6.2.3 Sensitivity to Memory Access Ratio r. The impact of memory-access ratio is investigated by varying γ from 0.0 to 1.0. In addition, we set $U_H(\Gamma_H) = 0.5$, $U_L(\Gamma_L) = 0.5$. The results are illustrated in Figure 7. AMC algorithm is insensitive to the memory-access ratio as AMC schedules the memory-access and computation phases sequentially, hence the schedulability ratio will not be impacted as long as the total workload remains unchanged. However, HEUR-DP-LO, HEUR-DP-HI and our proposed MCPA have higher schedulability ratio under more "balanced" memory-access and computation phases, this is because these three algorithms can take advantage of the parallel execution of different tasks' memory-access and computation phases.

However, when $r = 0$ and hence the tasks are memory-access only, both HEUR-DP-LO and HEUR-DP-HI are worse than our proposed MCPA; when $r = 1$, all the tasks are computation only, all the HEUR-DP-LO, HEUR-DP-HI, and our MCPA approaches are degraded to deadline monotonic algorithm, therefore, their performances converge to the same point.

6.2.4 Sensitivity to the Task Set Size $|\Gamma|$. In this set of experiments, we set $U_H(\Gamma_H) = 0.7$, $U_L(\Gamma_L) = 0.6$, and the memory access ratio $r = 0.6$. As 2BF is time unaffordable under large task set, we exclude it in this set of comparisons. The experimental results are illustrated in Figure 8. AMC still performs the worst, which is almost useless. HEUR-DP-LO, HEUR-DP-HI, and our proposed MCPA algorithms have higher schedulability ratio under larger task set—this is due to the following reason: under the same total utilization, larger task set implies that the same workload will be divided into more chunks and the priorities can be assigned at smaller granularity, and hence it is more apt to be schedulable.

7 CONCLUSION

In this article, we developed our approach for two-phase MC task set under fixed-priority scheduling. Different from traditional MC tasks having computation phase only, a new two-phase MC task model consisting of both memory-access and computation phases was proposed. Upon the

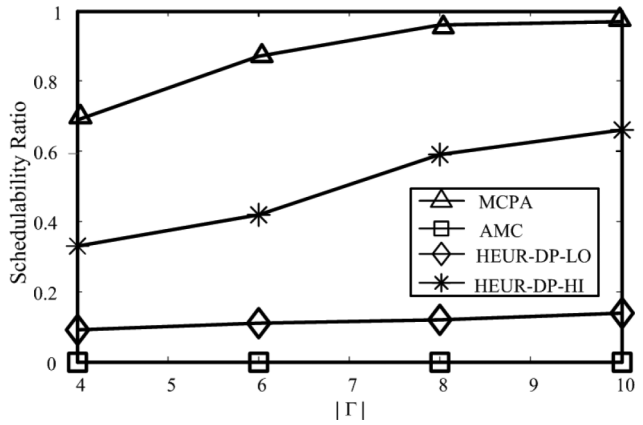

Fig. 8. Schedulability ratio comparison under varied task set size.

new task model, a fixed priority scheduling algorithm was developed in the following two steps: we first established the schedulability test under the given memory-access and computation priorities, and then devised a two-phase priority assignment strategy to find the best memory-access and computation priorities regarding to schedulability ratio. The experiment results revealed that existing scheduling theories could not be applied to the newly developed task model and our proposed two-phase priority assignment approach, which is not the optimal but performs much better than any existing algorithm.

In this article, the MC tasks were modeled with one memory-access phase and one computation phase, our immediate future work is to extend the current work for MC tasks with multiple memory-access and computation phases. In addition, we are building a many-core computing platform and plan to further evaluate the proposed models and approaches under real hardware platforms.

REFERENCES

N. C. Audsley. 2001. On priority assignment in fixed priority scheduling. *Inf. Process. Lett.* 79, 1 (May 2001), 39–44.

James Barhorst, Todd Belote, Pam Binns, Jon Hoffman, James Paunicka, Prakash Sarathy, John Scoredos, Peter Stanfill, Douglas Stuart, and Russel Urzi. 2009. Mcar white paper: A research agenda for mixed-criticality systems. In *CPS Week 2009 Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification 2009.*

S. Baruah, V. Bonifaci, G. D'Angelo, Haohan Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. 2012a. Scheduling real-time mixed-criticality jobs. *IEEE Trans. Comput.* 61, 8 (2012), 1140–1152.

S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. Van der Ster, and L. Stougie. 2012b. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS'12).* 145–154.

S. K. Baruah, A. Burns, and R. I. Davis. 2011. Response-time analysis for mixed criticality systems. In *Proceedings of the IEEE 32nd Real-Time Systems Symposium (RTSS'11).* 34–43.

S. Baruah and B. Chattopadhyay. 2013. Response-time analysis of mixed criticality systems with pessimistic frequency specification. In *Proceedings of the IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'13).* 237–246.

S. Baruah and S. Vestal. 2008. Schedulability analysis of sporadic tasks with multiple criticality specifications. In *Proceedings of the Euromicro 2008 Conference on Real-Time Systems (ECRTS'08).* 147–155.

Enrico Bini and Giorgio C. Buttazzo. 2005. Measuring the performance of schedulability tests. *Real-Time Systems* 30, 1–2 (2005), 129–154.

A. Burns and R. I. Davis. 2013. *Mixed Criticality Systems: A Review.* Technical Report MCC-1(b). Department of Computer Science, University of York, East Lansing, MI.

G. C. Buttazzo, G. Lipari, and L. Abeni. 1998. Elastic task model for adaptive rate control. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS'98)*. 286–295.

D. de Niz, K. Lakshmanan, and R. Rajkumar. 2009. On the scheduling of mixed-criticality real-time task sets. In *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS'09)*. 291–300.

P. Ekberg and Wang Yi. 2012. Bounding and shaping the demand of mixed-criticality sporadic tasks. In *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems (ECRTS'12)*. 135–144.

Pontus Ekberg and Wang Yi. 2014. Bounding and shaping the demand of generalized mixed-criticality sporadic task systems. *Real-Time Systems* 50, 1 (2014), 48–86.

Mathai Joseph and Paritosh K. Pandya. 1986. Finding response times in a real-time system. *Comput. J.* 29, 5 (1986), 390–395.

Wataru Kaneko, Kenji Kono, and Kentaro Shimizu. 2003. Preemptive resource management: Defending against resource monopolizing dos. *Applied Informatics*, vol 21, 662–669.

Z. Li, S. Ren, and G. Quan. 2014. Dynamic reservation-based mixed-criticality task set scheduling. In *High Performance Computing and Communications, Proceedings of the IEEE 6th International Symposium on Cyberspace Safety and Security, 2014 IEEE 11th International Conference on Embedded Software and Systems (HPCC, CSS, ICESS'14)*. 603–610.

Zheng Li and Li Wang. 2016. Memory-aware scheduling for mixed-criticality systems. In *Proceedings of the 16th International Conference on Computational Science and Its Applications (ICCSA'16)*. Springer International.

Z. Li, N. Wu, and M. Zhou. 2012. Deadlock control of automated manufacturing systems based on petri nets;a literature review. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42, 4 (July 2012), 437–462.

G. Lipari and G. C. Buttazzo. 2013. Resource reservation for mixed criticality systems. In *Proceedings of the Workshop on Real-Time Systems: The Past, the Present, and the Future*. 60–74.

C. L. Liu and James W. Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM* 20, 1 (Jan. 1973), 46–61.

D. Liu, J. Spasic, N. Guan, G. Chen, S. Liu, T. Stefanov, and W. Yi. 2016. EDF-VD scheduling of mixed-criticality systems with degraded quality guarantees. In *Proceedings of the 2016 IEEE Real-Time Systems Symposium (RTSS'16)*. 35–46.

Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio Buttazzo. 2015. Memory-processor co-scheduling in fixed priority systems. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems (RTNS'15)*. ACM, New York, NY, 87–96.

A. Melani, M. Bertogna, R. Davis, V. Bonifaci, A. Marchetti-Spaccamela, and G. Buttazzo. 2016. Exact response time analysis for fixed priority memory-processor co-scheduling. *IEEE Trans. Comput.* PP, 99 (2016), 1.

Taeju Park and Soontae Kim. 2011. Dynamic scheduling algorithm and its schedulability analysis for certifiable dual-criticality systems. In *Proceedings of the 9th ACM International Conference on Embedded Software (EMSOFT'11)*. ACM, New York, NY, 253–262.

Hang Su and Dakai Zhu. 2013. An elastic mixed-criticality task model and its scheduling algorithm. In *Proceedings of the 2013 Design, Automation Test in Europe Conference Exhibition (DATE'13)*. 147–152.

Steve Vestal. 2007. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS'07)*. 239–243.

Oreste Villa, Gianluca Palermo, and Cristina Silvano. 2008. Efficiency and scalability of barrier synchronization on NoC based many-core architectures. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'08)*. ACM, New York, NY, 81–90.

Qingling Zhao, Zonghua Gu, and Haibo Zeng. 2015. Resource synchronization and preemption thresholds within mixed-criticality scheduling. *ACM Trans. Embed. Comput. Syst.* 14, 4 (2015), 81.

Received September 2016; revised May 2017; accepted May 2017