PT-AMC: Integrating Preemption Thresholds into Mixed-Criticality Scheduling

Qingling Zhao, Zonghua Gu
College of Computer Science, Zhejiang University
Hangzhou, China

Haibo Zeng
Dept. of Electrical & Computer Engr., McGill University
Montreal, Canada

Abstract—**Mixed-Criticality Scheduling (MCS) is an effective approach to addressing diverse certification requirements of safety-critical systems that integrate multiple subsystems with different levels of criticality. Preemption Threshold Scheduling (PTS) is a well-known technique for controlling the degree of preemption, ranging from fully-preemptive to fully-non-preemptive scheduling. We present schedulability analysis algorithms to enable integration of PTS with MCS, in order to bring the rich benefits of PTS into MCS, including minimizing the application stack space requirement, reducing the number of runtime task preemptions, and improving schedulability.**

I. Introduction

Today's complex Cyber-Physical Systems often need to integrate multiple diverse applications with varying levels of *criticality*, or importance. For example, in the avionics certification standard DO-178B, there are 5 criticality levels, classified according to the degree of damage caused by failure of the application: *catastrophic; hazardous; major; minor; no effect.* Similarly, in the automotive safety certification standard ISO 26262, there are 4 criticality levels: *ASIL (Automotive Safety and Integrity Level) A-D*. In both avionics and automotive domains, there has been a trend of increasing degree of integrating multiple applications on shared hardware platforms, i.e., moving from many distributed small processors to a few centralized, large, and powerful processors. It is very challenging to integrate diverse applications with different criticality levels when they share the same hardware processor. Current industry practice achieves strong isolation by resource reservation and virtualization, e.g., the ARINC 653 software architecture for Integrated Modular Avionics, but this approach tends to result in design rigidity and hardware overbooking.

Mixed-Criticality Scheduling (MCS) [11] has been proposed to achieve strong temporal protection for high-criticality applications and efficient utilization of hardware resources. Although MCS was originally proposed in the context of safety-critical avionics applications, it is also finding its way into the automotive domain, which shares many of the same characteristics and trends with avionics applications. For example, it has been integrated into the tool-chain and real-time OS products from the well-known automotive software vendor SymtaVision, according a paper describing their design flow in the context of the automotive industry standard AUTOSAR [4]. As the automotive industry faces intense cost-cutting pressure in today's hyper-competitive market, it is important to minimize hardware costs by adopting cheaper processors with limited hardware (processing and memory) resources. (In contrast, the avionics industry is much less cost-sensitive, since airplanes are not mass-produced.) *In this paper, we address the problem of minimizing application task stack usage and improving schedulability in the context of MCS*, in order to cut hardware costs, which has not been addressed before, perhaps due to the perceived lack of motivation in the avionics domain.

There are two broad categories of OS scheduling algorithms: *preemptive* or *non-preemptive* scheduling. While conventional wisdom may indicate that preemptive scheduling can achieve better schedulability (also called feasibility) at the cost of increased implementation complexity and runtime overhead, it is in fact not true. It is well-known that schedulability of a taskset with non-preemptive scheduling does not imply schedulability with preemptive scheduling, and vice versa. That is, no one algorithm dominates the other in terms of schedulability consistently across all tasksets. [10] introduced the notion of *Preemption Threshold Scheduling (PTS)*, which had been integrated into a commercial real-time OS (ThreadX from Express Logic), and to some degree, supported by the automotive OSEK/AUTOSAR OS standard through the concept of *non-preemption groups*. PTS allows a task to disable preemption from higher priority tasks up to a specified threshold priority; only tasks with priorities higher than the given task's threshold are allowed to preempt it. Its benefits include: minimizing the application stack space requirement compared to fully-preemptive scheduling, which is very important for mass-produced, cost-sensitive embedded systems; reducing the number of runtime task preemptions compared to preemptive scheduling; improving schedulability (with proper setting of priorities and preemption thresholds) compared to both preemptive and non-preemptive scheduling.

In this paper, we integrate PTS and MCS by introducing the preemption threshold mechanism into MCS, in the context of *Fixed-Priority scheduling* and *Adaptive Mixed-Criticality (AMC)* variant of MCS. The resulting integrated algorithm is denoted as *PT-AMC*. We present a sufficient schedulability test for PT-AMC by computing the *Worst-Case Response Time* (WCRT) of each task. The test is then used within a priority and preemption threshold assignment algorithm to improve schedulability and reduce stack space usage.

II. Overview of PT-AMC

We consider a set of N independent and sporadic tasks $\Gamma = \{\tau_1, \tau_2, ..., \tau_N\}$. Each task τ_i has a tuple of parameters $< T_i, D_i, \vec{C_i}, L_i >$ where T_i is its period; D_i is its deadline; L_i is criticality level; $\vec{C_i}$ is a vector of computation values — one per criticality level, with the constraint that $L1 > L2 \Rightarrow C_i(L1) \geq C_i(L2)$ for any two criticality levels L1 and L2 of task τ_i .

In accordance with the current literature on MCS, we assume that (a) tasks are independent (i.e., there is no blocking due to shared resources), (b) tasks do not suspend themselves, other than at the end of their computation, (c) the overheads due to context switching, etc., are negligible (i.e., assumed to be zero), (d) the system is *dual-criticality*, i.e., there are only two criticality levels: *HI* (high) and *LO* (low), (e) tasks have implicit deadlines, i.e., $D_i = T_i$.

According to PTS, each task τ_i is assigned a *priority* $\pi_i \in [1, N]$ and a *preemption threshold* $\gamma_i \in [\pi_i, N]$. These values are assigned off-line and remain constant at run-time. We assume that larger numbers denote higher priority, and priority assignment is unique, i.e., no two tasks have the same priority. When a task is released, it is inserted into the ready queue with its priority π_i. When the task is dispatched at runtime, its priority is effectively raised to its preemption threshold γ_i and it keeps this priority until the execution is finished. In other words, the currently-executing task τ_i can only be preempted by another task τ_j if $\pi_j > \gamma_i$. PTS can be viewed as a technique for controlling the degree of preemption, with fully-preemptive and fully-non-preemptive scheduling as its special cases. If each task's preemption threshold is set to be equal to its priority, i.e., $\gamma_i == \pi_i$, then PTS becomes preemptive scheduling; if each task's preemption threshold is set to infinity, i.e., $\gamma_i == \infty$, or equivalently, equal to the highest priority in the taskset, then PTS becomes non-preemptive scheduling.

[2] introduced two variants of MCS if execution platform supports runtime monitoring and enforcement of task execution time limits: *Static Mixed-Criticality (SMC),* where a LO-critical task τ_l is aborted (de-scheduled) if it executes for more than its estimated execution time $C_l(LO)$; and *Adaptive Mixed-Criticality (AMC)*, where *all* LO-critical tasks are aborted if *any* job (from any task τ_a) executes for more than $C_a(LO)$. Since [2] proved that AMC dominates SMC in terms of schedulability, we adopt AMC in this paper. In AMC, dispatching of jobs for execution occurs according to the following rules (These rules are closely based on [2], with the additional elements from PTS introduced and depicted in **boldface**).

R1: There is a system-wide *criticality level indicator* Φ, initialized to LO. At any time instant, the system can be in one of two modes: *LO-critical mode* ($\Phi == LO$), and *HI-critical* mode ($\Phi == HI$).

R2: While ($\Phi == LO$), at each instant the waiting job generated by the task with highest priority that is **higher than the preemption threshold** of the currently-executing job is selected for execution.

R3: If the currently-executing job executes for its LO-critical WCET without signaling completion, then the system goes into HI-critical mode, i.e., $\Phi \leftarrow HI$.

R4: Once ($\Phi == HI$), jobs with criticality level LO will not be executing. Henceforth, at each instant the waiting job generated by the HI-critical task with the highest priority that is **higher than the preemption threshold** of the currently-executing job is selected for execution.

For AMC, it is important to distinguish between the *criticality level* of each task (LO or HI), denoted by the terms *LO-critical task* and *HI-critical task*; and the system-wide *criticality level indicator* Φ, denoted by the terms *LO-critical mode* and *HI-critical mode*. Each task's criticality level is fixed at design time, while the system can transition from LO-critical mode to HI-critical mode at runtime (but not vice versa). For clarity, we adopt the convention of using superscripts *LO* or *HI* to label variables related to system-wide LO or HI critical modes, e.g., R_i^{LO} denotes the WCRT of task τ_i in LO-critical mode; and use *(LO)* or *(HI)* to label variables related to each task's criticality level, e.g., $C_i(LO)$ refers to the WCET estimate of task τ_i at LO-critical level (although τ_i itself may be either LO or HI-critical).

Table 1 summarizes the key notations. By convention, τ_i is assumed as the task under analysis. Without loss of generality, we assume $\tau_{i,0}$, the first job of τ_i, is released at time 0.

Table 1. Key notations used in this paper

$\tau_{i,q}$	The q-th job of task τ_i (index q starts from 0)
$hp(\tau_i)$	Set of tasks with higher priority than τ_i ($\pi_j > \pi_i$)
$hep(\tau_i)$	Set of tasks with priority higher than or equal to τ_i ($\pi_j \geq \pi_i$), including τ_i itself
$hpH(\tau_i)$	Set of HI-critical tasks with higher priority than τ_i
$hpL(\tau_i)$	Set of LO-critical tasks with higher priority than τ_i
$ht(\tau_i)$	Set of tasks with priorities higher than preemption threshold of τ_i ($\pi_j > \gamma_i$)
$htH(\tau_i)$	Set of HI-critical tasks with priorities higher than preemption threshold of τ_i
$htL(\tau_i)$	Set of LO-critical tasks with priorities higher than preemption threshold of τ_i

III. WCRT Analysis for PT-AMC

From [2], we need to check three conditions to verify schedulability of an AMC taskset:

1) Verifying schedulability of the LO-critical mode,
2) Verifying schedulability of the HI-critical mode,
3) Verifying schedulability of the criticality change phase.

The third condition is necessary as it cannot be deduced from the schedulability of the stable modes, a well-known result for real-time systems with multiple modes of operation. The first two conditions are relatively easy to adapt to PTS, while the third condition is more involved, since one needs to be careful about the different possible time instants when the system can change from LO-critical to HI-critical mode, and their effects on WCRT of tasks with preemption thresholds.

[2] presented two methods for sufficient schedulability analysis of an AMC taskset with fixed priority scheduling by computing each task's WCRT: *AMC-rtb* (for *response time*

bound), and *AMC-max*. We adopt AMC-rtb as the basis for our algorithm in this paper, and leave AMC-max as part of future work. First, we review *AMC-rtb* [2]. Let R_i^{LO}, R_i^{HI}, R_i^* denote WCRT values of τ_i under the afore-mentioned three conditions, respectively:

$$R_i^{LO} = C_i(LO) + \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{R_i^{LO}}{T_j} \right\rceil \times C_j(LO)$$

$$R_i^{HI} = C_i(HI) + \sum_{\tau_j \in hpH(\tau_i)} \left\lceil \frac{R_i^{HI}}{T_j} \right\rceil \times C_j(HI)$$

$$R_i^* = C_i(HI) + \sum_{\tau_j \in hpH(\tau_i)} \left\lceil \frac{R_i^*}{T_j} \right\rceil \times C_j(HI) + \sum_{\tau_k \in hpL(\tau_i)} \left\lceil \frac{R_i^{LO}}{T_k} \right\rceil \times C_k(LO)$$

We use a small example for illustration purposes. Consider the taskset Γ comprised of three tasks in Table 2:

Table 2. An Example Taskset

τ_i	L_i	$C_i(LO)$	$C_i(HI)$	D_i	T_i
τ_1	*LO*	6	6	23	23
τ_2	*HI*	10	31	49	49
τ_3	*HI*	8	9	72	72

We seek to determine whether τ_3 can be assigned with the lowest priority according to AMC-rtb.

$$R_3^{LO} = 8 + \left\lceil \frac{R_3^{LO}}{23} \right\rceil \cdot 6 + \left\lceil \frac{R_3^{LO}}{49} \right\rceil \cdot 10 \quad R_3^{HI} = 9 + \left\lceil \frac{R_3^{HI}}{49} \right\rceil \cdot 31$$

$$R_3^* = 9 + \left\lceil \frac{R_3^{LO}}{23} \right\rceil \cdot 6 + \left\lceil \frac{R_3^*}{49} \right\rceil \cdot 31$$

These recursive equations yield $R_3^{LO} = 30$, $R_3^{HI} = 40$, $R_3^* = 83$. Since τ_3's WCRT exceeds its deadline, it cannot be assigned the lowest priority. We can check the other 2 tasks similarly, and find that no task can be assigned the lowest priority and still meet its deadline ($R_2^{LO} = 30$, $R_2^{HI} = 40$, $R_2^* = 52$; $R_1^{LO} = 24$). This means that the particular taskset is unschedulable according to AMC-rtb.

Next, we will give the WCRT analysis for PT-AMC.

A. Schedulability of LO-Critical Mode

The following equations are based on [8] for PTS, which corrected an error in [10]. Recall that in the LO-critical mode, no task τ_k executes for more than $C_k(LO)$. The worst-case blocking B_i^{LO} that τ_i can experience in the LO-critical mode is from tasks with lower priority and higher preemption threshold:

$$B_i^{LO} = \max_{\pi_j < \pi_i \le \gamma_j} \{C_j(LO)\} \tag{1}$$

The worst-case start time $S_{i,q}^{LO}$ of the q-th instance $\tau_{i,q}$ of task τ_i in the LO-critical mode is:

$$S_{i,q}^{LO} = B_i^{LO} + q \cdot C_i(LO) + \sum_{\tau_j \in hp(\tau_i)} (1 + \left\lfloor \frac{S_{i,q}^{LO}}{T_j} \right\rfloor) \cdot C_j(LO) \tag{2}$$

The worst-case finish time $F_{i,q}^{LO}$ of the q-th instance $\tau_{i,q}$ of task τ_i in the LO-critical mode is:

$$F_{i,q}^{LO} = S_{i,q}^{LO} + C_i(LO) + \sum_{\tau_j \in ht(\tau_i)} (\left\lceil \frac{F_{i,q}^{LO}}{T_j} \right\rceil - (1 + \left\lfloor \frac{S_{i,q}^{LO}}{T_j} \right\rfloor)) C_j(LO) \tag{3}$$

According to the commonly-accepted definition, during the *Level-i Busy Period* for τ_i, all instances of τ_i and higher priority tasks that arrive within the busy period must also finish within the busy period. The length of the level-i busy period in the LO-critical mode (denoted as LBP_i^{LO}) is given by the following recurrence relation:

$$LBP_i^{LO} = B_i^{LO} + \sum_{\tau_j \in hep(\tau_i)} \left\lceil \frac{LBP_i^{LO}}{T_j} \right\rceil C_j(LO) \tag{4}$$

τ_i's WCRT is the largest response time of all its jobs that arrive and finish within the busy period $[0, LBP_i^{LO}]$. Index of the last job within the busy period is $Q_i^{LO} = \lfloor LBP_i^{LO} / T_i \rfloor$, thus τ_i's WCRT in the LO-critical mode is:

$$R_i^{LO} = \max_{q \in \{0,1,\ldots Q_i^{LO}\}} (F_{i,q}^{LO} - q \cdot T_i) \tag{5}$$

The schedulability condition is:

$$\forall \tau_i \in \Gamma, R_i^{LO} \le D_i \tag{6}$$

Consider the example in Table 2 again: assuming τ_3 is assigned with the lowest priority and highest preemption threshold in the taskset, let's compute its WCRT in the LO-critical mode. We have $lp(\tau_3) = \varnothing$, $hp(\tau_3) = \{\tau_1, \tau_2\}$, $hep(\tau_3) = \{\tau_1, \tau_2, \tau_3\}$, $ht(\tau_3) = \varnothing$. τ_3's blocking time is 0 as the lowest-priority task.

$$LBP_3^{LO} = 0 + \left\lceil \frac{LBP_3^{LO}}{23} \right\rceil \cdot 6 + \left\lceil \frac{LBP_3^{LO}}{49} \right\rceil \cdot 10 + \left\lceil \frac{LBP_3^{LO}}{72} \right\rceil \cdot 8$$

$$Q_3^{LO} = \lfloor LBP_3^{LO} / 72 \rfloor$$

Hence $LBP_3^{LO} = 30$, $Q_3^{LO} = 0$, and

$$S_{3,0}^{LO} = (1 + \left\lfloor \frac{S_{3,0}^{LO}}{23} \right\rfloor) \cdot 6 + (1 + \left\lfloor \frac{S_{3,0}^{LO}}{49} \right\rfloor) \cdot 10$$

$$F_{3,0}^{LO} = S_{3,0}^{LO} + 8$$

Therefore, $S_{3,0}^{LO} = 16$, $R_3^{LO} = F_{3,0}^{LO} = 24$.

B. Schedulability of HI-Critical Mode

In the HI-critical mode, all LO-critical tasks are stopped, and only HI-critical tasks can execute. Hence the analysis is almost identical to the case of LO-critical mode, except that we only need to consider HI-critical tasks by replacing all occurrences of *LO* with *HI* in the above equations.

C. Schedulability of Criticality Change Phase

Recall the AMC algorithm: the system is initially in the LO mode; the mode change from LO mode to HI mode is triggered by some job $\tau_{k,n}$ executing for more than its WCET in LO-critical mode $C_k(LO)$. After the criticality change, all LO-critical tasks are stopped, so we no longer care about their schedulability, and we only need to check schedulability of HI-critical tasks in this case [2]. Without loss of generality, assume criticality change occurs at some time instant *s* during the invocation period of the q_s -th job τ_{i,q_s} of task τ_i , $s \in [q_s \cdot T_i, (q_s+1) \cdot T_i]$. We use S_{i,q_s}^{LO} to denote the worst-case (latest) start time of τ_{i,q_s} , and F_{i,q_s}^{LO} to denote the worst-case finish time of τ_{i,q_s} , when job τ_{i,q_s} starts and finishes *within the LO-critical mode*. For the sake of schedulability analysis of criticality change phase, we can further impose the upper bound $s \le F_{i,q_s}^{LO}$, i.e., $s \in [q_s \cdot T_i, F_{i,q_s}^{LO}]$, since if criticality change occurs after F_{i,q_s}^{LO} , τ_{i,q_s} will have finished execution within the LO mode and not be affected by it, so the WCRT analysis for LO-critical mode (presented in Section III.A) is sufficient to determine its schedulability.

Note that the system may experience a criticality change event and go from LO-critical mode to HI-critical mode either before or after job τ_{i,q_s} starts execution. If that happens, the actual worst-case start time and finish time (denoted as S_{i,q_s}^* and F_{i,q_s}^*) at runtime may be different from (larger or smaller than) S_{i,q_s}^{LO} and F_{i,q_s}^{LO} . However, S_{i,q_s}^{LO} and F_{i,q_s}^{LO} are still useful as time markers when we try to compute S_{i,q_s}^* and F_{i,q_s}^* . We divide the following analysis into two cases: criticality change occurs at or before S_{i,q_s}^{LO} ($s \in [q_s \cdot T_i, S_{i,q_s}^{LO}]$); or after S_{i,q_s}^{LO} , but at or before F_{i,q_s}^{LO} ($s \in (S_{i,q_s}^{LO}, F_{i,q_s}^{LO}]$), as shown in Fig. 1. In the former case, job τ_{i,q_s} executes its entirety in HI-critical mode; in the latter case, job τ_{i,q_s} executes its first part in LO-critical mode, but its latter part in HI-critical mode. Next, we derive WCRT analysis equations for these two cases separately.

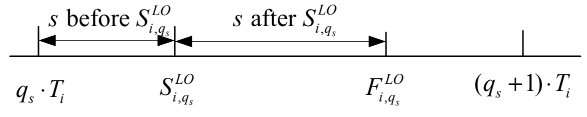

Figure 1. The system criticality change can occur either before or after S_{i,q_s}^{LO} .

Case 1: Criticality change occurs at or before S_{i,q_s}^{LO}

The worst-case start time S_{i,q_s}^* of τ_{i,q_s} when criticality change occurs before τ_{i,q_s} starts is constructed from multiple sources of interference it can experience, as shown in (7)

$$S^*_{i,q_s} = B^*_{i,q_s} + q_s \cdot C_i(LO) + I_{i,q_s}(LO) + I_{i,q_s}(HI) \qquad (7)$$

where B^*_{i,q_s} denotes the worst-case blocking time; $q_s \cdot C_i(LO)$ denotes the cumulative execution time of the sequence of jobs $\tau_{i,0}$ to τ_{i,q_s-1} ; $I_{i,q_s}(LO)$ and $I_{i,q_s}(HI)$ denote worst-case interferences from $hpH(\tau_i)$ and $hpL(\tau_i)$, respectively, before τ_{i,q_s} starts to execute (please refer to Table 1 for the definitions of $hpH(\tau_i)$ and $hpL(\tau_i)$).

According to the definition of busy period, the blocking job must finish before $\tau_{i,0}$ starts to run. Since we assume the criticality change occurs at time $s \le S_{i,q_s}^{LO}$, we must also have $s \le S_{i,q_s}^*$, despite the fact that S_{i,q_s}^* may be different from (either larger or smaller than) S_{i,q_s}^{LO} . If $q_s = 0$, the blocking job, which finishes before $\tau_{i,0}$ starts running, may finish *before* or *after* time *s*, i.e., in either LO-critical or HI-critical mode; if $q_s > 0$, the blocking job must finish *before* time *s*, i.e., in LO-critical mode. Hence the worst-case blocking time B^*_{i,q_s} is:

$$B^*_{i,q_s} = \begin{cases} \max(B_i^{LO}, B_i^{HI}), & q_s = 0 \\ B_i^{LO}, & q_s > 0 \end{cases}$$

Since all LO-critical tasks are prevented from executing after time instant *s* (inclusive), $s \le S_{i,q_s}^{LO}$, the worst-case interference caused by them can be upper-bounded by:

$$I_{i,q_s}(LO) = \sum_{\tau_j \in hpL(\tau_i)} \left\lceil \frac{S_{i,q_s}^{LO}}{T_j} \right\rceil \cdot C_j(LO) \qquad (8)$$

Since HI-critical tasks may interfere τ_{i,q_s} during the whole interval $[0, S_{i,q_s}^*]$, the worst-case interference caused by them is:

$$I_{i,q_s}(HI) = \sum_{\tau_j \in hpH(\tau_i)} (1 + \left\lfloor \frac{S_{i,q_s}^*}{T_j} \right\rfloor) \cdot C_j(HI) \qquad (9)$$

Therefore,

$$S_{i,q_s}^* = B^*_{i,q_s} + q_s \cdot C_i(LO) + \sum_{\tau_j \in hpL(\tau_i)} \left\lceil \frac{S_{i,q_s}^{LO}}{T_j} \right\rceil \cdot C_j(LO) + \sum_{\tau_j \in hpH(\tau_i)} (1 + \left\lfloor \frac{S_{i,q_s}^*}{T_j} \right\rfloor) \cdot C_j(HI) \qquad (10)$$

After τ_{i,q_s} starts to execute, it can only be preempted by tasks in $htH(\tau_i)$. All jobs of tasks in $htH(\tau_i)$ released before S_{i,q_s}^* must have completed, so the worst-case finish time F_{i,q_s}^* is:

$$F_{i,q_s}^* = S_{i,q_s}^* + C_i(HI) + \sum_{\tau_j \in htH(\tau_i)} (\left\lceil \frac{F_{i,q_s}^*}{T_j} \right\rceil - (1 + \left\lfloor \frac{S_{i,q_s}^*}{T_j} \right\rfloor)) \cdot C_j(HI) \qquad (11)$$

Case 2: Criticality change occurs after S_{i,q_s}^{LO}

To distinguish between Case 1 and Case 2, we label intermediate variables with an additional prime in the following discussions, e.g., $B^{*'}_{i,q_s}$, $S^{*'}_{i,q_s}$, $I'_{i,q_s}(LO)$, $I'_{i,q_s}(HI)$, $F^{*'}_{i,q_s}$. Since the system is initially in the LO-critical mode before the time of criticality change s, the worst-case blocking time $B^{*'}_{i,q_s}$ and worst-case start time $S^{*'}_{i,q_s}$ of τ_i are equal to those in the LO-critical mode:

$$B^{*'}_{i,q_s} = B_i^{LO}\ ; S^{*'}_{i,q_s} = S^{LO}_{i,q_s} \qquad (12)$$

The worst-case finish time $F^{*'}_{i,q_s}$ of τ_{i,q_s} is constructed from multiple forms of interference it can experience:

$$F^{*'}_{i,q_s} = S^{LO}_{i,q_s} + C_i(HI) + I'_{i,q_s}(LO) + I'_{i,q_s}(HI) \qquad (13)$$

where $I'_{i,q_s}(LO)$ and $I'_{i,q_s}(HI)$ are the worst-case interferences from tasks in $htL(\tau_i)$ and $htH(\tau_i)$, respectively, after τ_{i,q_s} starts to execute.

First, we consider LO-critical tasks. From the rules of AMC, all LO-critical tasks are prevented from executing after $s(S^{LO}_{i,q_s} < s \le F^{LO}_{i,q_s})$; by Equation (12), the actual worst-case start time $S^{*'}_{i,q_s} = S^{LO}_{i,q_s}$. Hence only the LO-critical, higher-priority tasks that arrive during time interval $[S^{LO}_{i,q_s}, F^{LO}_{i,q_s})$ can cause interference to τ_{i,q_s}, which is upper-bounded by:

$$I'_{i,q_s}(LO) = \sum_{\tau_j \in htL(\tau_i)} \left(\left\lceil \frac{F^{LO}_{i,q_s}}{T_j} \right\rceil - \left(1 + \left\lfloor \frac{S^{LO}_{i,q_s}}{T_j} \right\rfloor \right) \right) \cdot C_j(LO) \qquad (14)$$

Next, we consider HI-critical tasks. The HI-critical, higher-priority tasks that arrive during the time interval $[S^{LO}_{i,q_s}, F^{*'}_{i,q_s})$ can cause interference to τ_{i,q_s}, hence their worst-case interference time is upper-bounded by:

$$I'_{i,q_s}(HI) = \sum_{\tau_j \in htH(\tau_i)} \left(\left\lceil \frac{F^{*'}_{i,q_s}}{T_j} \right\rceil - \left(1 + \left\lfloor \frac{S^{LO}_{i,q_s}}{T_j} \right\rfloor \right) \right) \cdot C_j(HI) \qquad (15)$$

Therefore the worst-case finish time $F^{*'}_{i,q_s}$ is:

$$F^{*'}_{i,q_s} = S^{LO}_{i,q_s} + C_i(HI) + \sum_{\tau_j \in htL(\tau_i)} \left(\left\lceil \frac{F^{LO}_{i,q_s}}{T_j} \right\rceil - \left(1 + \left\lfloor \frac{S^{LO}_{i,q_s}}{T_j} \right\rfloor \right) \right) C_j(LO)$$
$$+ \sum_{\tau_j \in htH(\tau_i)} \left(\left\lceil \frac{F^{*'}_{i,q_s}}{T_j} \right\rceil - \left(1 + \left\lfloor \frac{S^{LO}_{i,q_s}}{T_j} \right\rfloor \right) \right) C_j(HI) \qquad (16)$$

Combining Equations (11) and (16), the worst-case finish time of τ_{i,q_s} is thus $\max(F^{*}_{i,q_s}, F^{*'}_{i,q_s})$.

Next, we determine the range of q_s to be considered for analysis during criticality mode change. Since the system criticality change from LO to HI may occur during any of the all possible jobs of τ_i within the busy period in the LO-critical mode, index q_s should range from 0 to $\lfloor LBP_i^{LO} / T_i \rfloor$. Note that even though the actual busy period at runtime in the presence of criticality mode change may be different from (larger or smaller than) LBP_i^{LO}, the possible time instants of criticality mode change still must fall within LBP_i^{LO}. If not, the criticality change occurs after τ_i's busy period has finished, and will not affect τ_i's WCRT.

Finally, τ_i's WCRT R_i^* can be determined as $R_i^* = \max_{q_s \in \{0 \ldots \lfloor LBP_i^{LO}/T_i \rfloor\}} (\max(F^{*}_{i,q_s}, F^{*'}_{i,q_s}) \text{-} q_s \cdot T_i)$.

D. Sufficient Schedulability Test for PT-AMC

Summarizing the results from previous subsections, the sufficient (but not necessary) schedulability test for PT-AMC is:

$$\begin{Bmatrix} \forall \tau_i \in \Gamma,\ R_i^{LO} \le D_i \\ \forall \tau_i \in H(\Gamma):\ \max(R^*_{i,}, R^{HI}_{i,}) \le D_i \end{Bmatrix}$$

IV. Experimental Evaluation

For the experiments, we adopt the branch-and-bound algorithm for setting task priorities and preemption thresholds in [11, 5], invoking the schedulability analysis algorithm for PT-AMC presented in this paper as a subroutine. The branch-and-bound algorithm was shown to be optimal, i.e., it always finds the feasible preemption threshold (PT) assignment if one exists, and then raises the PT assignments to yield the smallest possible total stack space size. This algorithm has worst-case exponential complexity, but runs reasonably fast for our tasksets. (If scalability becomes an issue for larger tasksets, the heuristic algorithm in [9] can be adopted instead, which was shown to be able to achieve near-optimal results.) We compare PT-AMC to other approaches from the literature, including:

- *AMC-rtb* and *AMC-max*: Schedulability analysis from [2], where priorities are assigned using Audsley's algorithm [1].
- *OCBP-prio*: The test for Own Criticality-Based Priority (OCBP) scheduling algorithm from [6], which is based on whether a feasible priority ordering can be found for all jobs in a busy period.
- *Vestal:* Schedulability analysis from [11]. The scheduling algorithm is equivalent to the SMC algorithm from [2].
- *OCBP-load*: The test from [7] for OCBP-based scheduling based on *load* of the taskset.

We generate synthetic tasksets based on the methodology in [6, 3]. A random taskset is generated by starting with an empty task set $\Gamma = \varnothing$, and successively adding random tasks into it. The generation of a random task is controlled by four parameters: the probability P_{HI} of a task with HI-criticality; the maximum ratio R_{HI} of HI- to LO-criticality WCET; the maximum low-criticality WCET value $C_{LO}^{\max}$; and the maximum period $T^{\max}$.

Following [3], we define the average utilization $U_{avg}(\Gamma)$ of a task set Γ as $U_{avg}(\Gamma) \overset{def}{=} (U_{LO}(\Gamma) + U_{HI}(\Gamma))/2$, where

$U_{LO}(\Gamma) \overset{def}{=} \sum_{\tau_i \in \Gamma} C_i(LO)/T_i; U_{HI}(\Gamma) \overset{def}{=} \sum_{\tau_i \in H(\Gamma)} C_i(HI)/T_i$. For a user-defined target average utilization U^*, each taskset is generated within the range of [U^*-0.005, U^*+0.005]. In our experiments, the target average utilization varies from 0.017 to 0.983. For each target utilization value, 1000 tasksets are randomly generated. Fig. 2 shows the acceptance ratio (fraction of schedulable task sets) as a function of target average utilization for tasksets generated using parameters P_{HI} =0.5, R_{HI} =4, $C_{LO}^{\max}$ =10, and $T^{\max}$ =200. As expected, the acceptance ratio of PT-AMC is better than most existing approaches. Specifically, when the taskset average utilization is in [0.717, 0.817], the acceptance ratio of PT-AMC can be larger than that of AMC-rtb or AMC-max by 8.7%-13.2%. Since OCBP-prio is not a static-priority scheduling algorithm, it is not surprising that it sometimes outperforms PT-AMC.

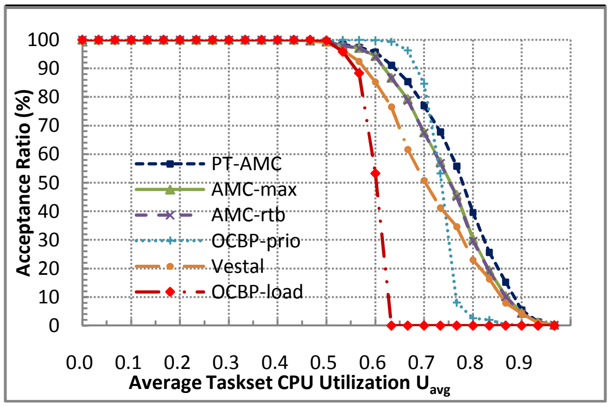

Figure 2. Percentage of schedulable tasksets vs. average CPU utilization U_{avg}. (Larger is better.)

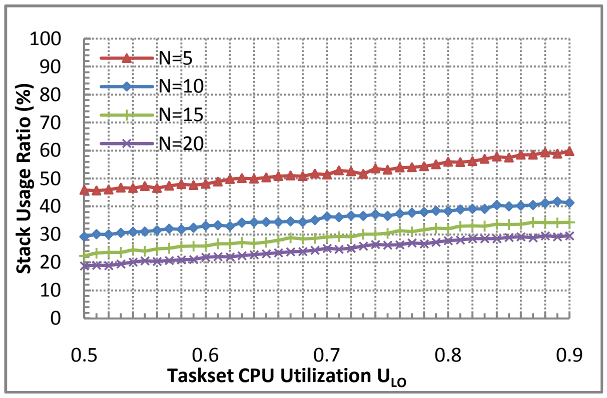

Figure 3. Stack usage ratio vs. LO-critical mode CPU utilization U_{lo}. (Smaller is better.)

For evaluating stack usage reduction, we slightly modify the synthetic taskset generator to have precise control over the number of tasks in a taskset, which has a large impact on total stack usage. Given the desired number of tasks N in the taskset Γ, and target system utilization $U_{LO}(\Gamma)$ computed from the $C_i(LO)$ values, tasksets are generated using the *UUnifast* algorithm as in [2]. Each task's stack space size is chosen from a uniform distribution over $\{S^{\min}, S^{\min}+1, ..., S^{\max}\}$, where $S^{\min}=20, S^{\max}=120$. As the performance metric, we define the *stack usage ratio* $\Omega(\Gamma)$ for taskset Γ as $\Omega(\Gamma) = Stack_{PT-AMC}(\Gamma) / Stack_{AMC}(\Gamma) \times 100\%$. We consider systems with N=5, 10, 15, 20 number of tasks. For each N, and each system utilization $U_{LO}(\Gamma)$ at the LO-critical mode from 50% to 90% (with a granularity of 1%), 1000 systems are randomly generated such that they are schedulable with the condition from AMC-rtb (for fully preemptive policy). All these 41000 systems are also schedulable with PT-AMC. In addition, PT-AMC can achieve significant savings in stack size, especially for larger number of tasks (Fig. 3). For example, it only uses 18%-30% of the original un-optimized stack space from AMC-rtb when N= 20.

V. Conclusions and Future Work

In this paper, we propose schedulability analysis techniques for PT-AMC, the first integration of preemption-threshold scheduling with mixed-criticality scheduling. For future work, we plan to address other variants of MCS, especially those based on EDF scheduling [3], which has been shown to achieve much better schedulability.

VI. Acknowledgements

We would like to thank Nan Guan and Pontus Ekberg for helpful discussions. This work is partially supported by NSFC Project Grant #61070002; National 863 Program Grant # 2011AA010104; NSERC Discovery Grant RGPIN 418741-12.

References

[1] N. Audsley, On priority assignment in fixed priority scheduling. *Information Processing Letters*, 79(1):39–44, 2001.

[2] S. Baruah, A. Burns, and R. Davis. Response-Time Analysis for Mixed Criticality Systems. In *Proc. IEEE Real-Time Systems Symposium*, 2011.

[3] P. Ekberg and W. Yi. Bounding and Shaping the Demand of Mixed-Criticality Sporadic Tasks. In *Proc. 24th Euromicro Conference on Real-Time Systems*, 2012.

[4] C. Ficek, K. Richter, and N. Feiertag. Schedule Design to Guarantee Freedom of Interference in Mixed Criticality Systems. In *Society of Automotive Engineers World Congress*, 2012.

[5] R. Ghattas and A. G. Dean. Preemption Threshold Scheduling: Stack Optimality, Enhancements and Analysis. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium*, 2007.

[6] N. Guan, P. Ekberg, M. Stigge, and W. Yi. Effective and Efficient Scheduling of Certifiable Mixed-Criticality Sporadic Task Systems. In *Proc. IEEE Real-Time Systems Symposium*, 2011.

[7] H. Li and S. Baruah. An Algorithm for Scheduling Certifiable Mixed-Criticality Sporadic Task Systems. In *Proc. IEEE Real-Time Systems Symposium*, 2010.

[8] J. Regehr. Scheduling tasks with mixed preemption relations for robustness to timing faults, In *Proc. IEEE Real-Time Systems Symposium*, 2002.

[9] M. Saksena and Y. Wang. Scalable Real-Time System Design using Preemption Thresholds. In *Proc. IEEE Real-Time Systems Symposium*, 2000.

[10] Y. Wang and M. Saksena. Scheduling Fixed-Priority Tasks with Preemption Threshold. In *Proc. IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 1999.

[11] S. Vestal. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. In *Proc. IEEE Real-Time Systems Symposium*, 2007.