Run and Be Safe: Mixed-Criticality Scheduling with Temporary Processor Speedup

Pengcheng Huang, Pratyush Kumar, Georgia Giannopoulou, Lothar Thiele
Computer Engineering and Networks Laboratory, ETH Zurich, 8092 Zurich, Switzerland
firstname.lastname@tik.ee.ethz.ch

Abstract—**Mixed-Criticality systems are arising due to the push from several major industries including avionics and automotive, where functionalities with different safety criticality levels are integrated into a modern computing platform to reduce size, weight and energy. The state-of-the-art research has focused on protecting critical tasks under the threat of task overrun, which is achieved by terminating less critical tasks or degrading their services to free system resources to guarantee critical tasks. We take in this paper a different approach to protecting critical tasks by embracing rich features of modern computing platforms. In particular, we explore dynamic processor speedup to aid the scheduling of mixed-criticality systems. We show that speedup in situation of overrun can not only help to protect the timeliness of critical tasks, but also to improve the degraded services for less critical tasks. Furthermore, we show that speedup is even more attractive as it can help the system to recover faster to normal operation. Thus, speedup could only be temporarily required and incur low cost. The proposed techniques are validated by both theoretical analysis and experimental results with an industrial flight management system and extensive simulations.**

I. INTRODUCTION

Today, a common trend in many industry sectors (e.g. automotive and avionics) is to consolidate tasks with varying *safety* requirements into a common computing platform [1], due to size, weight and power (SWaP) concerns. In parallel, modern computing platforms used to host mixed-criticality systems are becoming increasingly complex to attain high performance and rich features [2]. The crossroads of both trends poses enormous challenges on the design of mixed-criticality systems, where the ultimate goal is to provide heterogeneous safety assurances to tasks of different criticality levels.

One of such challenges is in the real-time domain [1]. With the adoption of modern computing platforms, accurate worst-case execution time (WCET) is becoming very hard to obtain [3], due to complex pipeline/memory architectures and resource contention patterns. For safety-oriented mixed-criticality systems, uncertainties in WCET, i.e. runtime overrun, should then be assumed and addressed when scheduling the system. Furthermore, tasks of different criticality levels should react to task overrun to different extents.

State-of-the-art. To date, the main thread of research on mixed-criticality systems has focused on the real-time aspect and a common model exists to specify such systems [4]–[6]. The system starts with the normal operation mode, where all tasks are guaranteed assuming no task overrun. However, if tasks overrun their expected WCETs, the system transits to a critical operation mode, during which the protection of critical tasks is commonly achieved by terminating less critical tasks [4], [5], or lately, by degrading the services [6] provided to less critical tasks. As a result, critical tasks may still meet deadlines by using the freed resources from less critical tasks.

Motivation. Degrading the services for less critical tasks or terminating them in the extreme can incur great performance/service loss. In addition, it has been shown recently that task terminating could actually violate system safety [7]. This is often overlooked for mixed-criticality systems: Tasks of each criticality level are associated with certain safety requirement, see e.g. the common safety regulation IEC 61508 [8]. While it is true that the safety requirement on a higher criticality level is more stringent, removing tasks of a lower criticality level simply removes the required safety associated to that criticality level. Therefore, for the sake of both safety and performance, alternative mechanisms for protecting critical tasks are needed.

Contribution. This paper explores the combination of mixed-criticality scheduling with architecture features of modern computing platforms. Our key insight is that, routine features of such platforms like dynamic voltage and frequency scaling (DVFS) [9] and dynamic cache partitioning and locking (DCPL) [10] could be used to facilitate the design of adaptive mixed-criticality systems. As a proof of concept, we solve in this paper mixed-criticality scheduling with the aid of DVFS. DVFS is conventionally used to conserve energy by exploring free slacks in the system to slow the processor down (underclocking) [9]. In contrast, we adopt DVFS to speed up the processor (overclocking) when there is a timing urgency caused by task overrun, such that all tasks could still meet their deadlines. Or in case service degradation for less critical tasks is permitted under overrun, speeding up the processor can improve the degraded services for less critical tasks. Notice that a similar concept already exists in the literature [11]. However, the approach therein addresses energy saving while we focus on real-time guarantees in this paper.

In practice, processor speedup increases energy dissipation and is often regulated by power/thermal management [12]. For example, Intel turbo boost technology would allow a maximum of 2x speedup for around 30s [12]. Moreover, fast recovery from adversary events like task overrun is often desired, especially for embedded systems deployed in safety-related domains. In both regards, we show that processor speedup can actually help the system to recover/reset faster: By speeding up the processor, the system overload could be resolved faster, enabling quick resuming of normal operation/speed. Nevertheless, in an extreme case when tasks overrun frequently and the system needs to overclock long, then we could monitor at runtime for how long the overclocking lasts. If this exceeds the time allowed, we could then terminate tasks instead of overclocking to reset the system to normal speed.

In the remainder of this paper, we adopt earliest deadline first (EDF) scheduling and provide theoretical results on: (1) calculating offline a minimum processor speedup to guarantee system schedulability in critical operation mode (Section III), and (2) quantifying offline the system resetting time under processor speedup (Section IV). In addition, we analytically show the tradeoffs between different design parameters including processor speedup, service degradation and resetting time in Section V. To demonstrate the applicability of our proposed techniques, we finally conduct experiments with both an industrial flight management system and extensive simulations on synthesized task sets (Section VI). Our results confirm that dynamic processor speedup can greatly increase system schedulability regions and is typically required for only short periods of time.

II. System model

We assume a set of n sporadic dual-criticality tasks $\tau = \{\tau_1, \cdots, \tau_i, \cdots, \tau_n\}$ scheduled on a uniprocessor. Each task may issue an infinite number of jobs. Task τ_i is characterized by the minimum inter-arrival time between consecutive jobs T_i, relative deadline D_i, and criticality level χ_i. We assume tasks have constrained deadlines, i.e. $\forall \tau_i \in \tau, D_i \leq T_i$. Furthermore, a task can either have high (HI) criticality or low (LO) criticality, i.e. $\forall \tau_i \in \tau, \chi_i \in \{\text{HI}, \text{LO}\}$. Moreover, we denote all χ criticality tasks as τ_χ, i.e. $\tau_\chi = \{\tau_i | \chi_i = \chi\}$.

To model the uncertainties in task WCETs, we adopt a common assumption in the literature [4]–[6]: The WCETs of tasks are modeled on all existing criticality levels, with the WCETs on HI criticality being more pessimistic than those on LO criticality (for safety assurance). We denote the WCET of τ_i on criticality χ as $C_i(\chi)$. In addition, we assume that LO criticality tasks are not allowed to exceed their LO criticality WCETs. Based on this model, the conventional mixed-criticality scheduling techniques [1] provide dynamic guarantees to all tasks, which can be specified by a simple mode switch protocol:

- The system starts with LO mode, where all tasks are guaranteed to meet their deadlines if they do not exceed their LO criticality WCETs.
- Whenever any HI criticality task exceeds its LO criticality WCET, the system transits immediately to HI mode, and all LO criticality tasks are dropped or their services are degraded hereafter to protect the timeliness of HI criticality tasks.

For each LO criticality task τ_i, the degraded service is interpreted as the increased inter-arrival time T_i and/or deadline D_i. For notational convenience, we denote the parameters of task τ_i in mode χ as: $\{T_i(\chi), D_i(\chi), C_i(\chi)\}$. Furthermore, for task parameters of HI criticality tasks, further *constraints* need to be applied for the reason explained as follows.

Preparation for overrun. For any HI criticality task τ_i, assume that it has the same deadline (original deadline D_i) in both LO and HI operation modes: $D_i(\text{HI}) = D_i(\text{LO}) = D_i$. Consequently, in LO mode, if τ_i exceeds its LO criticality WCET just at its deadline, it would then be impossible for the task to finish the increased load by its deadline. In light of this, HI criticality tasks need to finish before their actual deadlines in LO mode. This can be achieved by artificially shortening their deadlines in LO mode to "prepare" for task overrun, see e.g. [4], [5]. Therefore, $\forall \tau_i \in \tau_{\text{HI}}$, it would be necessary to set $D_i(\text{LO}) < D_i(\text{HI})$.

Based on our model and assumptions, $\forall \tau_i \in \tau_{\text{HI}}$:

$$\begin{aligned} T_i(\text{HI}) = T_i(\text{LO}) = T_i, D_i(\text{LO}) < D_i(\text{HI}) = D_i, \\ C_i(\text{HI}) \geq C_i(\text{LO}), \end{aligned} \tag{1}$$

and $\forall \tau_i \in \tau_{\text{LO}}$:

$$\begin{aligned} T_i(\text{HI}) \geq T_i(\text{LO}) = T_i, D_i(\text{HI}) \geq D_i(\text{LO}) = D_i, \\ C_i(\text{HI}) = C_i(\text{LO}). \end{aligned} \tag{2}$$

Notice that the above notations embrace task terminating as a special case – if LO criticality tasks are terminated, then:

$$T_i(\text{HI}) = +\infty, D_i(\text{HI}) = +\infty, \ \forall \tau_i \in \tau_{\text{LO}}. \tag{3}$$

Other notations. For convenience of presentation, we extend the semantics of the conventional integer operator mod: $a \bmod b = a - \left\lfloor \frac{a}{b} \right\rfloor \cdot b$, where a and b can now be real numbers.

III. Minimum Processor Speedup to handle overrun

We present in this section how to compute the minimum required processor speedup to handle task overrun and guarantee task deadlines. For this purpose, we will first revisit some results on system schedulability analysis.

Since EDF is assumed in this paper, we could use demand bound analysis to determine the system schedulability. The demand bound function for task τ_i in a time interval of length Δ is defined as the worst-case total execution times of all τ_i's jobs with both arrival times and deadlines in this time interval. Known results exist in case that task parameters are constant, e.g. in LO mode, the demand bound function of any task can be calculated as [5]:

$$\text{DBF}_{\text{LO}}(\tau_i, \Delta) = \max\{\left\lfloor \frac{\Delta - D_i(\text{LO})}{T_i(\text{LO})} + 1 \right\rfloor, 0\} \times C_i(\text{LO}). \tag{4}$$

To guarantee *schedulability in LO mode*, it then suffices when the sum of demand bound functions for all tasks is no greater than the provided processing resource in any time interval [5].

The schedulability analysis in HI mode is non-trivial, since there could exist schedulability dependencies between LO and HI modes. This is because unfinished jobs in LO mode are forced to take the processing resources in HI mode to meet their HI mode deadlines. In light of this, their effects need to be taken into account when calculating the demand bound functions in HI mode. We adopt in this paper a known result to calculate such HI mode demand bound functions [5], [6]:

Lemma 1. *For any task $\tau_i \in \tau$, define a set of functions as follows:*

$$w(\tau_i, \Delta) = (\Delta \bmod T_i(HI)) - (D_i(HI) - D_i(LO)), \tag{5}$$

$$r(\tau_i, \Delta, w(\cdot)) = \begin{cases} \min\{w(\tau_i, \Delta), C_i(LO)\} & \textit{if } w(\tau_i, \Delta) \geq 0 \\ \quad +C_i(HI) - C_i(LO), & \\ 0, & \textit{otherwise} \end{cases}. \tag{6}$$

The demand bound function of τ_i in HI mode can be calculated as:

$$DBF_{HI}(\tau_i, \Delta) = r(\tau_i, \Delta, w(\cdot)) + \left\lfloor \frac{\Delta}{T_i(HI)} \right\rfloor \cdot C_i(HI). \tag{7}$$

Notice that in Lemma 1, $r(\tau_i, \Delta, w(\cdot))$ essentially quantifies the impacts of unfinished jobs in LO mode on HI mode system demands. Furthermore, we see that HI mode system demands depend on: (1) the preparation for task overrun in LO mode, i.e. LO mode deadlines of HI criticality tasks, and (2) the degraded services in HI mode, i.e. new deadlines and job inter-arrival times for LO criticality tasks.

Based on the above calculations of demand bound functions, we can now compute a minimum processor speedup to *guarantee schedulability in HI mode*.

TABLE I: Example task set

τ	χ	C_i(LO)	C_i(HI)	D_i(LO)	D_i(HI)	T_i(LO)	T_i(HI)
τ_1	HI	2	7	4	10	12	12
τ_2	LO	3	3	6	6	10	10

Theorem 2. *Assume the processor speeds up by a factor s when entering HI mode to handle overrun. The minimum processor speedup factor, $s_{\min}$, which guarantees HI mode schedulability, can be calculated as follows:*

$$s_{\min} = \max_{\Delta \geq 0} \left\{ \sum_{\tau_i \in \tau} DBF_{HI}(\tau_i, \Delta) \Big/ \Delta \right\}. \tag{8}$$

Proof: All proofs in this paper can be found in [13]. ■

Notice that in (8), we allow the divisor Δ to be zero: If the total HI mode system demands are non-zero in a interval of zero length, the system will require infinite speedup in HI mode, i.e. $s_{\min} = +\infty$. This could actually happen if the deadlines of HI criticality tasks are not shortened in LO mode, see our discussions regarding (1).

Example 1. *Consider a simple dual-criticality task set with parameters shown in Table I. The LO criticality task τ_2 here needs to adhere to its original service parameters in HI mode. With dynamic processor speedup allowed, we can apply (8) and calculate the minimum required speedup for HI mode is $\frac{4}{3}$. Furthermore, if we allow τ_2 to degrade its service in HI mode by setting $D_2(HI) = 15, T_2(HI) = 20$, then the required speedup factor can be reduced to 0.875. In this case, the system can actually slow down in HI mode despite the fact that τ_1 overruns, since the system load is greatly reduced by degrading the services for τ_2. We plot in Figure 1 our results for both cases. As we can see, the computed minimum speedup factors do guarantee HI mode schedulability.*

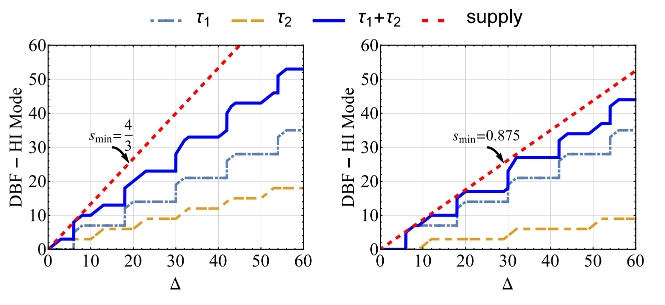
Computation efficiency. (8) suggests that the computation of the minimum processor speedup would require examination of all non-negative interval lengths. However, in practice, this computation can be done efficiently in pseudo-polynomial time due to the periodicity and piecewise-linearity of the demand bound function (7). For space reasons, we refer the interested readers to [13] for more details.

Runtime. During runtime, whenever some HI criticality task exceeds its LO criticality WCET, the system transits to HI mode and the processor is speeded up by $s_{\min}$. Our offline bound of $s_{\min}$ ensures that all tasks will meet their deadlines whenever the system enters HI mode.

IV. Service Resetting Time under Processor Speedup

We proceed to show that speeding up the processor to handle task overrun is even more attractive, as it can help the system to recover normal operation/speed faster. Together with the fact that task overrun is rare, this suggests processor speedup can incur low cost as it would only be temporarily required.

To quantify service resetting time under processor speedup, we need to first identify a sufficient condition under which a system can be safely reset to LO mode. Since we do not know statically how long the system will overrun, we can safely assume that the system can recover when the processor is idle, i.e. when all arrived workloads are finished. Notice that, there could be infinitely many idling points in HI mode. Thus, one needs to find the closest processor idling time after entering HI mode, at which point the system can safely recover.



(a) No service degradation (b) Service degradation

Fig. 1: Minimum speedup and demand bound functions

As a second step, we need to bound the worst-case arrived system loads/demands *starting* from the transition to HI mode, as this is required to upper-bound when in the nearest future can all the arrived demands be finished. Special attention needs to be paid to unfinished jobs in LO mode, since those "partial" jobs can be viewed as if they are released at the time of mode transition and must finish in HI mode. In fact, there is already one known result [6] in the literature which solves this problem. However, the technique proposed therein relies essentially on exhaustive search to find the worst-case arrived demands after entering HI mode. In the following, we will analytically identify such worst-case scenario.

For presentation convenience, we assume the system transits to HI mode at time $\hat{t}$. In addition, let us denote the start and end of a time interval of length Δ as t^{Δ}_{start} and t^{Δ}_{end}, resp. Furthermore, we denote the *latest* jobs of τ_i arriving no later than t^{Δ}_{start} and t^{Δ}_{end} as μ and λ, resp. The arrival time of these two jobs are denoted as t^{μ}_{a}, t^{λ}_{a}, resp. Figure 2 depicts graphically our notations.

Lemma 3. *For a time interval of length Δ starting from the transition to HI mode ($t^{\Delta}_{start} = \hat{t}$), the worst-case arrived demand for any task $\tau_i \in \tau$ is obtained when this time interval ends with a future job arrival of τ_i, i.e. $t^{\Delta}_{end} = t^{\lambda}_{a}$.*

The essential idea to prove Lemma 3 is to show that, by letting $t^{\Delta}_{end} = t^{\lambda}_{a}$ (shift the dotted time interval in Figure 2 to the left), the released demands of all jobs including μ and λ are not decreased. Again, we refer the interested readers to [13] for detailed explanations. Now, based on Lemma 3, we can quantify the worst-case arrived demand for any task starting from the transition to HI mode and bound when the system can be safely recovered.

Theorem 4. *Assume the processor speeds up by a factor s when entering HI mode at time $\hat{t}$, and define function $w'(\cdot)$ as follows:*

$$w'(\tau_i, \Delta) = (\Delta \bmod T_i(HI)) - (T_i(HI) - D_i(LO)). \tag{9}$$

The worst-case arrived demand bound of τ_i in time interval $[\hat{t}, \hat{t} + \Delta]$ can be calculated as:

$$ADB_{HI}(\tau_i, \Delta) = r(\tau_i, \Delta, w'(\cdot)) + \left(\left\lfloor \frac{\Delta}{T_i(HI)} \right\rfloor + 1\right) \cdot C_i(HI), \tag{10}$$

where function $r(\cdot)$ is defined in (6).

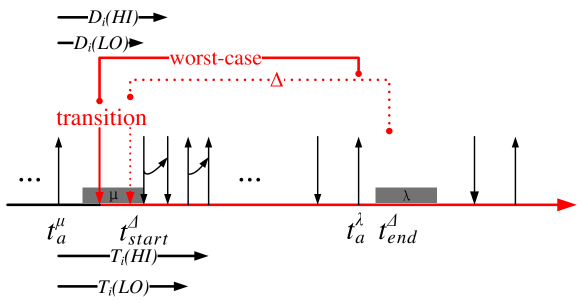


Fig. 2: Worst-case scenario for arrived demand

If the system is idle at time $\hat{t} + \Delta_R$, it must follow that:

$$\sum_{\tau_i \in \tau} ADB_{HI}(\tau_i, \Delta_R) \le s \cdot \Delta_R. \tag{11}$$

With the calculation of worst-case arrived demands starting from $\hat{t}$ in (10), the system is idle whenever the arrived demands are no greater than the supplied resources, as stated in (11). Since the system can potentially have many idling points in HI mode, we can then choose the first idling point after mode transition, the time distance between which and the transition point gives a safe lower-bound on the service resetting time. Formally, this can be formulated as follows.

Corollary 5. *If the processor speed is increased by a speedup factor s after transiting to HI mode, a safe service resetting time can then be lower-bounded as:*

$$\Delta_R = \min\{\Delta \ge 0 : \sum_{\tau_i \in \tau} ADB_{HI}(\tau_i, \Delta) \le s \cdot \Delta\}. \tag{12}$$

Example 2. *Consider the same task set as shown in Table I. According to (12), we can calculate the service resetting time for this task set. Our results are depicted in Figure 3.*

If no service degradation is allowed, we calculate that the service resetting time is 17.25 when s equals $\frac{4}{3}$. In addition, if s is increased to 2, then the service resetting time can be reduced to 6 since now overload is resolved faster with higher processor speed. This is shown in Figure 3a.

The trend of service resetting time as s increases is better illustrated with Figure 3b. As convinced by the results, there is a clear gain if the dynamic processor speedup is increased. We will present how to derive closed-formulas to express this relation in Section V. Furthermore, if service degradation is enabled in parallel to processor speedup, the service resetting time can be further reduced as less overload needs to be addressed. For our results here, the degraded service parameters as shown in Example 1 are adopted.

Computation efficiency. Similar to the computation of the minimum processor speedup, the computation problem here can also be reduced to pseudo-polynomial complexity. Again, for details, we refer the interested readers to [13].

Runtime. During runtime, whenever the processor is idle, the system switches back to LO mode, and the processor speed is restored to its original speed. Our offline bound of resetting time predicts when such restoration can happen.

Remark. Our calculation of the service resetting time Δ_R does not assume any particular task overrun pattern. If we assume in the worst-case two bursts of overrun are separated by at least T_O units of time ($T_O \gg 0$ since overrunning expected WCET is rare), then the frequency that the system needs to speedup is bounded by $\frac{1}{T_O}$ as long as $\Delta_R \le T_O$. In typical cases, the time used to speedup (Δ_R) is often limited to a few tens of seconds due to power/thermal management, which would anyway satisfy this constraint taking $\Delta_R \ll T_O$.

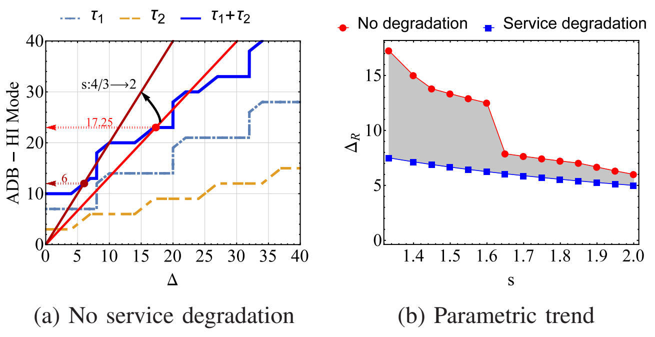


Fig. 3: Service resetting time under dynamic processor speedup

V. Special Case and System Level Trade-offs

To theoretically show the relation between different system parameters, the current research on mixed-criticality often adopts implicit-deadline tasks with two assumptions [4], [6]: (1) Deadlines of HI criticality tasks in LO mode are shortened by a common factor $0 < x < 1$ to prepare for overrun:

$$D_i(\text{LO}) = x \cdot D_i(\text{HI}), T_i(\text{HI}) = T_i(\text{LO}) = D_i(\text{HI}), \forall \tau_i \in \tau_{\text{HI}}. \tag{13}$$

(2) Deadlines of LO criticality tasks are scaled up by a common factor $y \ge 1$ in HI mode to react to overrun:

$$D_i(\text{HI}) = y \cdot D_i(\text{LO}), T_i(\chi) = D_i(\chi), \forall \tau_i \in \tau_{\text{LO}}\ \forall \chi. \tag{14}$$

We continue to show that with those assumptions, closed-formula solutions can be derived for computing the minimum required processor speedup and the service resetting time. In the following, we denote $\frac{C_i(\chi)}{T_i(\chi)}$ as $U_i(\chi)$, where $\chi \in \{\text{HI}, \text{LO}\}$.

A. Processor speedup

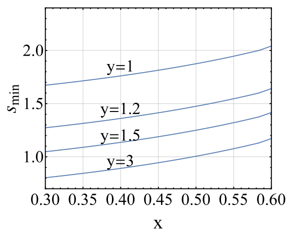
We first present result on bounding how overrun preparation x and service degradation y will affect the required speedup.

Lemma 6. *Given design parameters x and y, the minimum required speedup to guarantee HI mode schedulability can be upper-bounded as:*

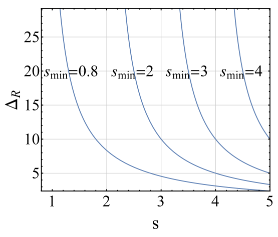
$$s_{\min} = \sum_{\tau_{HI}} \max \begin{cases} \frac{U_i(HI)}{U_i(LO)+(1-x)}, \\ \frac{U_i(HI)-U_i(LO)}{1-x} \end{cases} + \sum_{\tau_{LO}} \frac{U_i(LO)}{U_i(LO)+(y-1)}. \tag{15}$$

Lemma 6 clarifies that $s_{\min}$ will monotonically decrease with decreasing x and/or increasing y. We explain the intuitions behind this along with the following example.

Example 3. *Consider the task set shown in Table I, where task parameters are now modified according to (13) and (14). Based on Lemma 6, we depict in Figure 4a the impacts of overrun preparation x and service degradation y on the minimum required speedup to guarantee HI mode schedulability. As we can see, for smaller x (more overrun preparation in LO mode), the minimum required speedup is decreased. This is because in*



(a) Speedup



(b) Resetting time

Fig. 4: Various tradeoffs: Impact of overrun preparation x and service degradation y on required speedup and service resetting time

this case more resources are statically reserved for overrun by making HI criticality tasks finish earlier in LO mode. Thus, less speedup is required to address overload. In addition, we see that, with more service degradations (larger y), the required speedup also decreases. The reason is the same as explained for x, degrading the service for LO criticality tasks more will reduce HI mode system load, leading to less required speedup.

B. *Service resetting time*

We proceed to present how the service resetting time can be affected by system design parameters using a closed formula.

Lemma 7. *Given design parameters x, y and speedup s in HI mode, the service resetting time can be bounded as:*

$$\Delta_R = \frac{\sum_{\tau_i \in \tau} C_i(HI)}{s - s_{\min}}, \tag{16}$$

where $s_{\min}$ is given by (15).

With Lemma 7, we can see there is a clear gain in service resetting time when processor speedup in HI mode is increased. In addition, the service resetting time would be $+\infty$ if the minimum processor speedup $s_{\min}$ is chosen.

Example 4. *With the same task set used in previous examples, we plot in Figure 4b Δ_R against s with different minimum required system speedups. Essentially, $s_{\min}$ represents the system load in HI criticality mode. We observe, with artificially increased $s_{\min}$ (more system load in HI mode), the service resetting time is increased as it will take longer to resolve overload. For the same reason, similar trend can be observed if we decrease the actual system speedup s.*

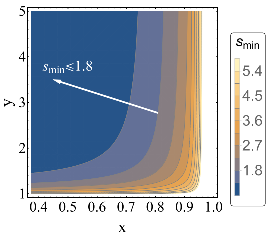
VI. Evaluation

We present in this section the validation of our proposed techniques by an industrial flight management system (FMS) and extensive simulations.

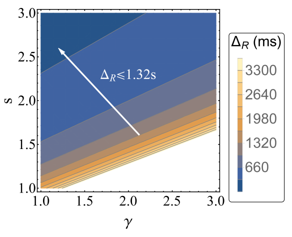
A. *Flight management system*

We adopt a subset of an industrial implementation of FMS, which consists of 7 DO-178B criticality level B (HI) and 4 criticality level C (LO) tasks. All tasks can be modeled as implicit deadline sporadic tasks, with task minimum inter-arrival times in the range of 100ms to 5s. For detailed parameters, we refer the readers to [6]. We present our results in Figure 5.

As suggested by Figure 5a, with decreasing x (better safety preparation) or increasing y (more service degradation), the required speedup in HI mode to meet deadlines is reduced, see (13) and (14) for definitions of x and y. This is because in both



(a) Contour - speedup



(b) Contour - resetting time

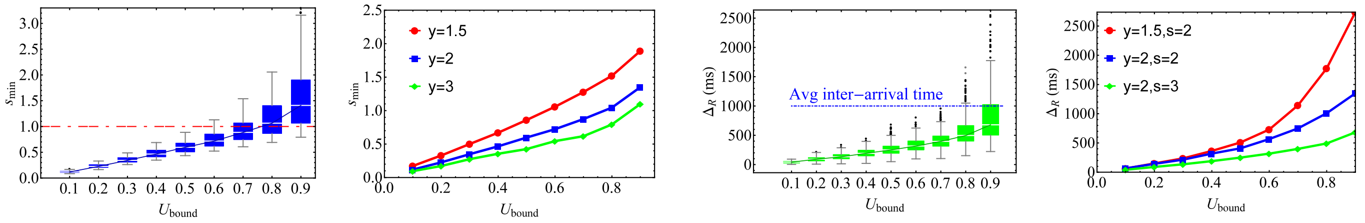
Fig. 5: Experimental results on FMS

cases, the system load in HI mode is decreased. The contour plot here clearly depicts the freedom in setting x and y for a certain speedup in HI mode. Furthermore, Figure 5b presents how the service resetting time is affected by the speedup in HI mode s and the uncertainty in HI criticality tasks' workloads γ ($\gamma := \frac{C_i(\text{HI})}{C_i(\text{LO})}, \forall \tau_i \in \tau_{\text{HI}}$). Our results show that with increasing γ or decreasing s, the service resetting time is increased, as increased overload is resolved more slowly. In addition, we observe that FMS takes in the worst-case less than 3s to recover with a speedup of 2, indicating that dynamic processor speedup could indeed only be temporarily required.

B. *Extensive simulations*

To show the applicability of our proposed techniques to general task sets, we now conduct extensive experiments on synthesized task sets. We adopt a random task generator as proposed in [4]. The task generator starts with an empty task set and continuously adds new random tasks to this set until certain system utilization U_{bound} is met. For details, we refer the readers to [4]. In the following, we generate 500 task sets at each system utilization point and conduct our experiments. Our results are shown in Figure 6, and we observe:

- Our proposed techniques can successfully bound the required processor speedup and the service resetting time in all tested cases. As the system utilization U_{bound} increases, both the required speedup and the service resetting time increase since more overload needs to be addressed in HI mode.
- The maximum required processor speedup is less than 3.3 for all tested cases ($U_{bound} = 0.9$ in Figure 6a). In this case, the 50th percentile value of processor speedup is only 1.4. In addition, for all cases when $U_{bound} \leq 0.5$, the maximum required speedup is less than 1, indicating that the system can even slow down in HI mode. Furthermore, we see that speedup indeed greatly improves system schedulability: Less than 25% task sets are schedulable when $U_{bound} = 0.9, s_{\min} = 1$, which is increased to 75% when $s_{\min} = 1.9$.
- The longest service resetting time for all tested cases is less than 2.6s ($U_{bound} = 0.9$ in Figure 6c), while the median value of resetting time in this case is only 678.6ms. Furthermore, we observe that the median/average service resetting time increases slowly with increased system utilization. However, the worst-case (dotted outliers in Figure 6c) can increase dramatically. In most test cases, the system can be reset within the average inter-arrival time of all tasks.
- With more service degradation, both the minimum required processor speedup and the system service resetting time can be reduced (increasing y for Figure 6b and Figure 6d), as less overload needs to be resolved in HI mode. Furthermore, with increasing processor speedup s in HI mode, the service resetting time can be further reduced, as convinced by the results in all test cases in Figure 6d.



(a) Boxwhisker - speedup (b) Degradation impact (c) Boxwhisker - resetting time (d) Speedup & degradation impact

Fig. 6: Simulation using synthesized task sets, with task minimum inter-arrival times randomly chosen from 2ms to 2s, task LO criticality utilization $C_i(\text{LO})/T_i(\text{LO})$ randomly chosen from 0.01 to 0.2 and $\gamma = C_i(\text{HI})/C_i(\text{LO})$ $(\forall \tau_i \in \tau_{\text{HI}})$ randomly chosen from 1 to 3. Figure 6a is obtained assuming $y = 2$ and shows the distribution of $s_{\min}$. Figure 6c is obtained assuming $y = 2, s = 3$ and shows the distribution of Δ_R. Figure 6b and Figure 6d show the median values of our results across different system utilizations. x in all cases is set to the minimum to guarantee LO mode schedulability [6].

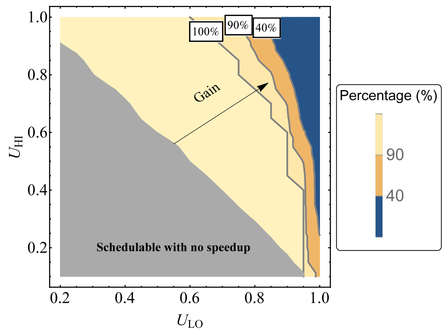


Fig. 7: Schedulability region under temporary processor speedup: 2x speedup for no longer than 5s. For this set of experiments, γ (defined in Figure 6) is set to 10 to cover more search spaces. All other task generation parameters are the same as used in Figure 6. $U_\chi = \sum_{\chi_i \geq \chi} C_i(\chi)/T_i(\chi), \chi \in \{\text{HI}, \text{LO}\}$.

C. Schedulability regions

We finally present in Figure 7 the schedulability regions under dynamic processor speedup. For the results in this figure, we set $s = 2$ and explicitly require that the resetting time must fulfill $\Delta_R \leq 5s$. We show how the system schedulability region could be increased with such temporary processor speedup. For this purpose, we generate in total 6.84×10^4 random task sets at all $(U_{\text{HI}}, U_{\text{LO}})$ points and conduct our experiments, see Figure 7 for explanation of U_χ. We assume LO criticality tasks are terminated in HI mode here. At each $(U_{\text{HI}}, U_{\text{LO}})$ point, we consider a small neighboring region to it $(U_\chi \pm 0.025)$, and compute the percentage the system is schedulable as the ratio of the number of schedulable task sets to the number of all task sets in this neighboring region.

Based on our results, we observe that: First, the region in which the system is 100% schedulable is greatly increased when compared to no processor speedup is used. Furthermore, at high system utilization points, temporary processor speedup can still guarantee a large portion of schedulable task sets. For example, when both U_{HI} and U_{LO} equal 0.85, 90% task sets generated here can still be successfully scheduled with 2x processor speedup for no longer than 5s.

VII. Conclusion

We propose in this paper to adopt dynamic processor speedup to handle task overrun and protect the timeliness of critical tasks for mixed-criticality systems. This approach is desirable as conventional approaches assume degrading less critical tasks or terminating them, which can incur great performance penalty or even lead to violation of system safety. We provide results on calculating the minimum speedup to satisfy real-time requirements in situation of overrun. Furthermore, we show that processor speedup can help the system to recover faster, which makes it even more attractive for mixed-criticality systems. Our proposed techniques are demonstrated with an industrial flight management system and extensive simulations, while various system level trade-offs are shown.

References

[1] A. Burns and R. Davis, "Mixed criticality systems-a review," 2014.

[2] B. D. de Dinechin, D. van Amstel, M. Poulhiès, and G. Lager, "Time-critical computing on a single-chip massively parallel processor," in *Proceedings of the Conference on Design, Automation & Test in Europe*, 2014, pp. 97:1–97:6.

[3] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem&mdash;overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, May 2008.

[4] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie, "The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems," in *ECRTS*, 2012, pp. 145–154.

[5] P. Ekberg and W. Yi, "Bounding and shaping the demand of generalized mixed-criticality sporadic task systems," *Real-Time Systems*, pp. 1–39, 2013.

[6] P. Huang, G. Giannopoulou, N. Stoimenov, and L. Thiele, "Service adaptions for mixed-criticality systems," in *ASP-DAC*, 2014, pp. 125–130.

[7] P. Huang, H. Yang, and L. Thiele, "On the scheduling of fault-tolerant mixed-criticality systems," in *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, ser. DAC '14, New York, NY, USA, 2014, pp. 131:1–131:6.

[8] S. Brown, "Overview of iec 61508. design of electrical/electronic/programmable electronic safety-related systems," *Computing & Control Engineering Journal*, 2000.

[9] L. Benini, A. Bogliolo, A. Paleologo, and G. De Micheli, "Policy optimization for dynamic power management," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 6, pp. 813–833, Jun 1999.

[10] I. Puaut and A. Arnaud, "Dynamic instruction cache locking in hard real-time systems," in *Proc. of the 14th Int. Conference on Real-Time and Network Systems*. Citeseer, 2006.

[11] P. Huang, P. Kumar, G. Giannopoulou, and L. Thiele, "Energy efficient dvfs scheduling for mixed-criticality systems," in *Proceedings of the 14th International Conference on Embedded Software*, ser. EMSOFT '14, 2014, pp. 11:1–11:10.

[12] A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann, "Power management architecture of the 2nd generation intel® core microarchitecture, formerly codenamed sandy bridge." Hotchips, 2011.

[13] P. Huang, P. Kumar, G. Giannopoulou, and L. Thiele, "Run and be safe: Mixed-criticality scheduling with temporary processor speedup," ETH Zurich, Laboratory TIK, Tech. Rep. 357, September 2014.