A Four-Mode Model for Efficient Fault-Tolerant Mixed-Criticality Systems

Zaid Al-bayati, Jonah Caplan, and Brett H. Meyer
McGill University, Montréal, Canada
{zaid.al-bayati@mail.,jonah.caplan@mail.,brett.meyer@}mcgill.ca

Haibo Zeng
Virginia Tech, Blacksburg, VA, USA
hbzeng@vt.edu

***Abstract*—Mixed-criticality systems (MCS) integrate components from different levels of criticality onto the same platform. MCS, like all other electronic systems, are susceptible to transient faults. These systems must mitigate the effects of faults and provide recovery mechanisms when faults occur. In this paper, we consider the problem of designing and scheduling certifiable fault-tolerant mixed-criticality systems. To address certification and transient faults, two-mode models must treat any single overrun or fault as a combination of the two, reserving time for the re-execution of tasks with extended execution time. We therefore propose a new four-mode model that addresses fault and execution time overrun with separate modes. This model, combined with the selective continuation of low-criticality tasks, improves the quality of service (QoS) to these tasks while providing the same guarantee to high-criticality tasks. Experimental results show that QoS improvements of up to 42.9% can be achieved by the new model. Furthermore, we show how the model and its schedulability analysis can be calibrated to realistic failures rates to achieve even more efficient designs.**

I. Introduction

Embedded systems today are performing more complex and diverse tasks than ever before, as cost constraints drive the integration of diverse features in highly integrated systems. Some of these features might be critical to the operation or the safety of the system, thus requiring explicit certification, while others do not. *Criticality* is defined as the level of assurance needed by a particular task in the system [6]. Systems composed of features with different criticalities are called *Mixed Criticality Systems* (MCS). MCS have applications in numerous domains such as automotive and avionics.

Most of the work on mixed-criticality scheduling assumes that the tasks in the system can be categorized into two levels of criticality: high-criticality (HI-criticality or simply HI) tasks that are safety-critical; and, low-criticality (LO-criticality or simply LO) tasks [6]. Designers want to schedule all tasks, while certification authorities require strict safety guidelines be met, but are only concerned with HI tasks. Consequently, certification authorities have more pessimistic assumptions about the worst-case execution time (WCET) of HI tasks than designers. Hence, a HI task τ_i has two WCET estimates, $C_i(LO)$ and $C_i(HI)$, in the LO and HI modes respectively, with $C_i(LO) \leq C_i(HI)$. All systems begin executing both HI and LO tasks (LO mode). According to the AMC (Adaptive Mixed Criticality) scheduling scheme [3], a transition to HI mode occurs when any task j in the system executes beyond $C_j(LO)$ (task overrun), and all LO tasks are dropped to make room for the increase in WCET budget for HI tasks.

To ensure safety, system operation must be guaranteed even when transient faults occur [2]. Transient faults require re-execution, while task overruns require task-dependent increases in execution budget. Researchers have begun to evaluate MCS in the context of transient faults, but often subsume overruns and task re-execution in response to transient error [10], [11]. This overlooks the fact that re-execution budgets and conservative WCET estimation are different, yet both must be considered. Other work differentiates between faults and overruns [15], but conservatively drops all LO tasks when either event occurs. Designers, however, are increasingly concerned with quality of service (QoS, the fraction of LO tasks scheduled in a given mode) [6], [18]. Hence, new techniques are needed to improve LO task QoS while satisfying certification and fault tolerance requirements.

In this paper, we propose a novel, four-mode model for MCS that addresses both certification and reliability requirements, while retaining as many LO tasks as possible when either overruns or transient faults occur. The proposed model differentiates between transient faults (TF mode), execution time overruns (OV mode), and their combination (HI mode). When fault-tolerant MCS are implemented in two modes as in [15], designers must assume transient fault and task overrun, though different in their rates of occurrence and consequences for task execution, always coincide. This results in overly pessimistic mode changes to HI mode. Our experimental results demonstrate that for single-core platforms, the new modes (TF, OV) of the four-mode system improve LO task QoS by 20.2% and 42.9% respectively compared to a two-mode model. Extended to partitioned multi-core systems, the benefits of the four-mode model are even more substantial: compared with a two-mode system, the new modes in our model improve LO task QoS by 43.9% and 91.6% on average.

We claim the following **contributions** in this paper:

(a) we propose a novel four-mode model to simultaneously consider certification, faults, and LO task QoS;
(b) we optimize LO task QoS by (i) differentiating requirements imposed by certification from those by fault tolerance, and (ii) reducing the provision for fault-induced re-execution through calibrating the model to realistic failure rates; and,
(c) we demonstrate our model on single core systems and partitioned multiprocessors, and illustrate the QoS improvement obtained using experiments with different settings.

II. Related Work

Research on mixed-criticality systems has flourished in recent years; interested readers are referred to a recent survey [6]. Earlier MCS scheduling work, such as AMC [3], addresses overruns by dropping all LO tasks in HI mode. This assumption is relaxed in [5] by allowing LO tasks to continue execution with relaxed parameters (for example, larger period). Later work such as [9] propose selecting a subset of LO tasks to continue execution with gradual degradation of service.

Other recent work has begun exploring the scheduling problem in the context of fault-tolerant MCS. For instance, in [13] re-execution slots are reserved for all tasks (both LO and HI). Since faults are rare events [8], such over-provisioning wastes CPU time and increases hardware costs. We start the system in a mode with no over-provisioning, and only consider re-execution for HI-criticality tasks and in response to faults. In [16], zonal and fault hazard analysis are used to constrain task allocation and re-execution and make offline guarantees for HI tasks. LO tasks are executed if the scenario allows it, but unlike our work, no guarantees are made. In [11], a reliability-aware mapping technique is proposed with worst case guarantees for MCS executing on a multiprocessor system-on-chip. However, WCET variation is assumed to be a result of the fault mitigation techniques such as re-execution. In [10], the reliability requirements of tasks are accounted for by deriving new values for the HI-criticality tasks' WCETs in the HI mode. None of the three works [10], [11], [16] model the basic MCS premise that HI tasks can have different WCETs coming from different sources, for example, the designer and the certification authority. The work in [15] addresses this issue and is similar to our work in considering both fault tolerance and certification requirements for mixed-criticality systems. However, it uses the standard two-mode model of MCS, and LO-criticality tasks are immediately dropped once either a fault or an overrun occurs, compromising QoS for LO tasks.

III. Assumptions and Notation

Our objective is to perform task allocation and scheduling for a set of sporadic mixed-criticality tasks $\Gamma = \{\tau_1, \tau_2, ...\}$. The supporting architecture can be single-core or multi-core. Tasks are scheduled with fixed priority, and, if the architecture is multicore, with partitioned scheduling. We assume that each core is capable of detecting faults, and core π is characterized by a failure rate λ_π (failures per unit time). Examples of such detection mechanism are acoustic sensors [17] or lockstep execution [2]. Scheduling on systems employing improvements to lockstep, such as dynamic core coupling [12] or on-demand redundancy [14], is the subject of future work.

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, migration, etc., are negligible. Each task τ_i has a 6-tuple of parameters $\langle L_i, C_i(LO), C_i(HI), T_i, D_i, \gamma_i \rangle$, where:

- $L_i \in \{\text{LO, HI}\}$ denotes the criticality level of τ_i.
- $C_i(LO)$ is the designer-specified WCET for τ_i.
- $C_i(HI)$ is the WCET by certification authorities.
- If $L_i = LO$, $C_i(HI) = C_i(LO)$, otherwise $C_i(LO) \leq C_i(HI)$.
- T_i denotes the period (minimum inter-arrival time) of τ_i; tasks have implicit deadlines, i.e., $D_i = T_i$.
- γ_i denotes the priority of τ_i.

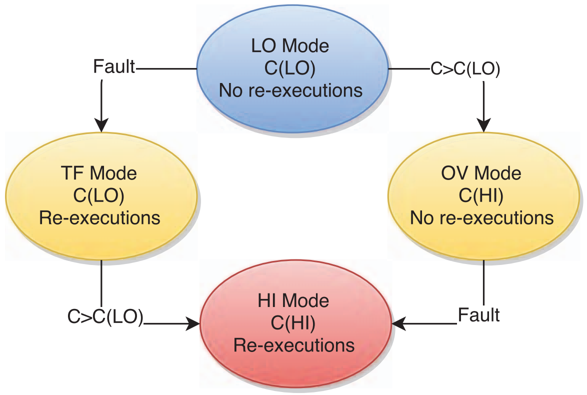


Fig. 1: The four-mode system model

u_i is the utilization of τ_i, where $u_i(L_i) = C_i(L_i)/T_i$.

Each criticality level is characterized by a maximum probability of failure to which each task of that level must conform, given as a probability of failure per hour (PFH). Sequential re-execution is used as the hardening mechanism to achieve the required reliability level. Each HI task τ_i must be guaranteed to be schedulable even when it re-executes in response to faults to achieve a failure rate of at most $PFH_i = PFH(HI)$. For LO-criticality tasks, we assume that they do not have specific PFH requirements, hence they do not need to be re-executed when faults occur. This work focuses on the more common transient faults affecting application tasks. Permanent hardware failures and operating system errors are out of the scope of this work.

IV. The Four-Mode System Model

To handle both task overruns and faults in the system efficiently, we propose a system model with four system modes for tasks with two criticality levels. In MCS, we cannot know *a priori* whether the task will actually overrun its $C(LO)$. These events are generally rare. Similarly, when considering faults, we cannot know in advance if the task will experience a transient error during its execution. If a task experiences a fault, it must be re-executed. These re-executions are also rare, and have a different effect on execution time.

Figure 1 illustrates the four system modes and possible mode changes. The modes are denoted LO (low), TF (transient faults), OV (task overruns), and HI (high). We denote the system mode by S, $S \in \{LO, TF, OV, HI\}$. The rationale for having four system modes, as opposed to the regular two-mode model of MCS in [3], [15], is to distinguish between these two different events (task overruns and transient faults) and provide suitable guarantees and quality of service in each case. From the original LO mode, each of these two independent events will cause the system to enter a different mode with sufficient guarantees (sufficient time for re-execution or overruns) for the corresponding type of event. To be safe, a fourth mode, HI, is used if an overrun occurs while the system is addressing a fault in TF, or vice versa. Most existing work on MCS perform forward mode changes only (from a lower mode to a higher mode) [6]. We adopt a similar strategy here, and defer reverse mode changes to future work.

A. Probability of Failure and Re-execution Requirements

Before we can perform scheduling, we must first derive the number of re-executions required to satisfy failure rate requirements. We denote the number of executions (including the initial one) required for a HI task τ_i to achieve a certain reliability (PFH) level in mode S as $n_i(S)$. The probability of failure per job of τ_i and the failure rate for a processor per τ_i's execution can be obtained by scaling PFH_i and λ_{π_i} respectively. Considering each execution of task τ_i as independent, then Constraint (1) must be satisfied for TF and HI modes. As a result, $n_i(S)$ is given by Equation (2).

$$PFH_i \cdot T_i \geq (\lambda_{\pi_i} \cdot C_i(S))^{n_i(S)} \tag{1}$$

$$n_i(S) = \begin{cases} \left\lceil \dfrac{\log(PFH_i \cdot T_i)}{\log(\lambda_{\pi_i} \cdot C_i(S))} \right\rceil, & S \in \{TF, HI\} \\ 1, & S \in \{LO, OV\} \end{cases} \tag{2}$$

B. System Operation

Our system operates as follows:

1) The system starts in the LO mode, where all tasks i execute once ($n_i(LO) = 1$) and can run up to $C_i(LO)$.
2) In LO mode, if any task i executes beyond $C_i(LO)$, the system moves into overrun (OV) mode where all HI-criticality tasks j can safely execute once ($n_j(OV) = 1$) up to $C_j(OV) = C_j(HI)$.
3) Alternatively, in LO mode, if any HI-criticality task experiences a transient fault, the system moves into transient fault (TF) mode where each HI-criticality task i is guaranteed a sufficient number of re-executions to satisfy their reliability requirements as per Eq. (2) ($n_i(TF) > 1$). *During the mode transition, the required re-executions for HI tasks must be guaranteed to finish before the deadline.* In this way, reliability requirements are met even when the task starts its execution in a mode that does not consider re-executions. The original task and all re-executions are allowed to execute up to $C_i(TF) = C_i(LO)$.
4) If any of the (re-)executions of a HI-criticality i task in TF mode overruns $C_i(LO)$, the system moves into HI mode. A move to this mode is also possible if the system is in OV mode and a transient fault occurs. In HI mode, HI-criticality tasks i can execute up to $C_i(HI)$ and re-execute $n_i(HI) \geq n_i(TF)$ times.

Modes OV and TF cover the basic cases of task overruns or transient faults by dropping some LO-criticality tasks to allow more time for HI tasks. Both events leading to HI have a low probability, however for highly critical systems, it is important to consider all cases. It is worth noting that in this mode, more re-executions for a HI-criticality task i might be needed than in TF mode ($n_i(HI) \geq n_i(TF)$), because HI-criticality tasks are assumed to run longer ($C_i(HI) \geq C_i(TF)$) and hence may experience more errors.

C. Providing QoS to LO-criticality tasks

As discussed in Section I, one of the criticisms the two-mode MCS model [3], [15] is the assumption that all LO tasks are dropped in higher modes [6]. Our proposed model does not drop all LO tasks in the modes OV, TF, and HI. Instead, LO-criticality tasks are selectively allowed to run in the new mode as long as this does not affect the schedulability of HI

TABLE I: Example Task Set

	$C(LO)$	$C(HI)$	$T = D$	L
τ_1	3	4	12	HI
τ_2	4	-	12	LO
τ_3	4	-	12	LO
τ_4	1	-	12	LO

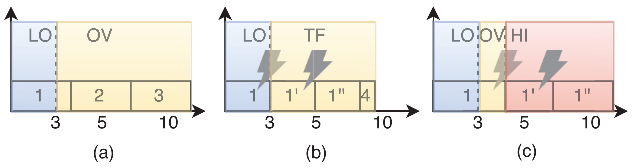


Fig. 2: An execution trace for the task set in Table I when (a) an overrun occurs (b) a fault occurs (c) both occur.

tasks. New schedulability analysis that extends AMC to the new model is presented in Section V. This analysis provides offline guarantees to all HI tasks and the selected subset of LO tasks. In this way, the designer can be sure of the schedulability of the task set regardless of the runtime scenario. We maximize the number of LO tasks scheduled in each mode; alternatively, the designer may select the LO tasks that continue.

D. An Illustrative Example

To illustrate the benefit of the proposed model, we apply the proposed model to the example task set in Table I. The tasks are arranged in priority order (the lowest indexed task has highest priority). We assume the number of executions for HI task τ_1 is $n_1(TF) = n_1(HI) = 3$. If an overrun occurs in τ_1 (Figure 2a), the system moves into OV mode where $12 - 4 = 8$ time units are available to run the LO-criticality tasks in the system. Therefore, we only need to drop one task (τ_4, for example). If τ_1 needs to re-execute due to faults (Figure 2b), since $n_1(TF) = 3$, the system can still schedule one LO task (τ_4). If both faults and overruns occur (Figure 2c), the system switches to HI mode, and τ_1 can still run safely with all re-executions. For a two-mode model, any overrun or fault will lead to a scenario similar to Figure 2c, and all LO tasks will be dropped immediately. The four-mode model clearly improves LO task QoS over the two-mode model.

V. Schedulability Analysis

A. Basic Analysis

We now present schedulability analysis for the four-mode system by extending the AMC-rtb approach [3] to the proposed new model. The AMC-max schedulability test [3] can be similarly extended and is left out due to space limitations.

For a system to be schedulable:

1) HI-criticality tasks must be scheduled in all four modes;
2) LO-criticality tasks must be scheduled in the LO mode.

Therefore, for modes OV, TF, and HI we only need to check the schedulability of HI tasks. It is not necessary for the schedulability of the system to guarantee the schedulability of any LO task in these modes. However, through design space exploration, we try to schedule LO tasks in these modes as long as the overall system schedulability is not affected.

For LO mode the response time of a task τ_i is given by:

$$R_i^{(LO)} = C_i(LO) + \sum_{j \in hp(i)} \left\lceil \frac{R_i^{(LO)}}{T_j} \right\rceil \cdot C_j(LO) \tag{3}$$

where $hp(i)$ is the set of tasks on τ_i's processor that have higher priority than τ_i (including both HI and LO tasks).

When evaluating schedulability in mode TF, we need to consider two cases. First, we need to consider the stable case: the given HI-criticality task itself and higher priority tasks. Second, we also need to verify that the mode change LO-to-TF does not cause the given task to miss its deadline. The response time of a task in the mode change is larger than in the stable mode, since in addition to the task itself and higher priority tasks executing in mode TF, we need to consider the effect of LO-criticality tasks that may be dropped in TF but have not finished executing. The worst case response time of a task in TF is hence the response time of the mode change and is given by Equation (4).

$$\begin{aligned} R_i^{(TF)} &= n_i(TF) \cdot C_i(LO) \\ &+ \sum_{j \in hpC(TF,i)} \left\lceil \frac{R_i^{(TF)}}{T_j} \right\rceil \cdot n_j(TF) \cdot C_j(LO) \\ &+ \sum_{k \in hp(i) - hpC(TF,i)} \left\lceil \frac{R_i^{(LO)}}{T_k} \right\rceil \cdot C_k(LO) \end{aligned} \tag{4}$$

In TF, we need to take into account task re-executions for HI-criticality tasks j, hence $C_j(LO)$ values are multiplied by the number of re-executions $n_j(TF)$. $hpC(TF, i)$ is the set of higher priority continuing tasks (i.e., tasks that continue to execute in the current mode TF). This set of tasks contains all HI tasks and those LO tasks selected to continue. The last summation considers the tasks that get dropped in the LO-to-TF mode change (in the set $hp(i) - hpC(TF, i)$). These tasks can only execute up to $R_i^{(LO)}$ since a mode change must occur before $R_i^{(LO)}$, otherwise τ_i would have terminated before the mode change. Since only LO tasks are allowed to be dropped, we use $C_k(LO)$ and $n_k = 1$ here.

The response time for mode OV can be similarly calculated. However, HI tasks can execute up to $C(HI)$ and no re-executions are considered ($n = 1$ for all tasks). The response time in mode OV is given by:

$$\begin{aligned} R_i^{(OV)} &= C_i(L_i) + \sum_{j \in hpC(OV,i)} \left\lceil \frac{R_i^{(OV)}}{T_j} \right\rceil \cdot C_j(L_j) \\ &+ \sum_{k \in hp(i) - hpC(OV,i)} \left\lceil \frac{R_i^{(LO)}}{T_k} \right\rceil \cdot C_k(LO) \end{aligned} \tag{5}$$

In mode HI, we need to consider both overruns ($C_i(HI)$) and re-execution ($n_i(HI) \geq 1$). Transitions to the HI mode can originate from either TF or OV mode. Both mode changes need to be verified. The worst case scenario occurs when we encounter two consecutive mode changes in quick succession (LO-to-TF-to-HI or LO-to-OV-to-HI).

Focusing on the LO-to-TF-to-HI mode change first (Equation (6)), the task τ_i under analysis and the high priority continuing tasks in mode HI $hpC(HI, i)$ are allowed to execute n times (first and second terms on the right hand side of the equation). For tasks that are allowed to continue in TF mode but dropped in HI mode (i.e., in the set $hpC(TF, i) - hpC(HI, i)$), we assume that the mode change from TF to HI has to happen before $R_i^{(TF)}$ (third term). Finally, for the remaining tasks in $hp(i)$ (i.e. tasks that started in LO and got dropped in TF), the interference is bounded by the fact that the mode change from LO to TF has to happen before $R_i^{(LO)}$ (last term). Note that we do not consider re-executions for those dropped tasks ($n_k = n_l = 1$) as they are all LO-critical.

$$\begin{aligned} R_i^{(HIa)} &= n_i(HI) \cdot C_i(L_i) \\ &+ \sum_{j \in hpC(HI,i)} \left\lceil \frac{R_i^{(HIa)}}{T_j} \right\rceil \cdot n_j(HI) \cdot C_j(L_j) \\ &+ \sum_{k \in hpC(TF,i) - hpC(HI,i)} \left\lceil \frac{R_i^{(TF)}}{T_k} \right\rceil \cdot C_k(LO) \\ &+ \sum_{l \in hp(i) - hpC(TF,i)} \left\lceil \frac{R_i^{(LO)}}{T_l} \right\rceil \cdot C_l(LO) \end{aligned} \tag{6}$$

Similarly, the task response time of τ_i for LO-to-OV-to-HI transition is presented in Equation (7):

$$\begin{aligned} R_i^{(HIb)} &= n_i(HI) \cdot C_i(L_i) \\ &+ \sum_{j \in hpC(HI,i)} \left\lceil \frac{R_i^{(HIb)}}{T_j} \right\rceil \cdot n_j(HI) \cdot C_j(L_j) \\ &+ \sum_{k \in hpC(OV,i) - hpC(HI,i)} \left\lceil \frac{R_i^{(OV)}}{T_k} \right\rceil \cdot C_k(LO) \\ &+ \sum_{l \in hp(i) - hpC(OV,i)} \left\lceil \frac{R_i^{(LO)}}{T_l} \right\rceil \cdot C_l(LO) \end{aligned} \tag{7}$$

The response time of τ_i in HI mode is the maximum of the two possible transitions:

$$R_i^{(HI)} = \max(R_i^{(HIa)}, R_i^{(HIb)}) \tag{8}$$

B. Reducing Model Pessimism

It is unlikely that all tasks would be required to re-execute the maximum number of times within a given period. Transient faults are rare, and the number of faults can be limited with a parameter provided by the designer [7], [15]. We relax the assumption that failures are independent, and let F be the maximum number of faults expected in any interval of length $D_{\max}$ where $D_{\max}$ is the largest relative deadline among all tasks. This parameter can be obtained through fault analysis considering the expected environmental conditions.

If we limit the maximum number of additional re-executions that need to be considered in the response time calculation F, it is possible to further improve LO-criticality task QoS. For example, consider the task set in Table II. Assuming tasks are sorted by priority order (task τ_1 has the highest priority). The response time of task τ_3 in mode TF

TABLE II: An Example Task Set

	$C(LO)$	$C(HI)$	$T=D$	n	L
τ_1	3	4	20	2	HI
τ_2	4	6	20	2	HI
τ_3	4	-	20	1	LO
τ_4	1	-	20	1	LO

assuming no tasks are dropped is $2 \times 3 + 2 \times 4 + 4 = 18$. If the designer knows that a maximum of one error can occur in 20 time units ($F = 1$), then only one job of either task τ_1 or τ_2 can fail within a single job of τ_3. The worst case occurs when τ_2 fails and the response time can be bounded at 14.

More generally, for modes TF and HI, the analysis can be formulated as an ILP (Integer Linear Programming) problem, by small modification to Eqs. (4), (6), and (7). We replace n_i (the number of re-executions required by τ_i to meet its PFH requirement) by $1+f_i$ where f_i is the number of re-executions we need to consider from τ_i in the worst case response time calculation. In the case of TF, we then maximize:

$$\begin{aligned} R_i^{(TF)} &= (1+f_i)\cdot C_i(LO) \\ &+ \sum_{j\in hpC(TF,i)} \left\lceil \frac{R_i^{(TF)}}{T_j} \right\rceil \cdot (1+f_j)\cdot C_j(LO) \\ &+ \sum_{k\in hp(i)-hpC(TF,i)} \left\lceil \frac{R_i^{(LO)}}{T_k} \right\rceil \cdot C_k(LO) \end{aligned} \tag{9}$$

under the constraints:

$$1+f_i \le n_i, \quad f_i \ge 0, \quad \forall \tau_i \tag{10a}$$

$$\sum_i f_i \le F. \tag{10b}$$

Similar formulations can be derived for the transitions to HI, Eqs. (6), and (7), respectively. In practice, the maximum can be easily determined with polynomial complexity by sorting the list of tasks with $n > 1$ by descending utilization, then adding the maximum number of re-executions for each task in the list until there are F errors.

VI. Experimental Results

We conducted experiments to compare the LO-criticality task QoS (the fraction of LO tasks scheduled in a given mode) achieved by our four-mode model with the traditional two-mode model in a variety of scenarios. We varied system utilization, the total number of tasks, and the fraction of HI tasks, for both single-core and lockstep multicore architectures.

By default, half of the tasks were HI-critical. The periods of tasks were randomly selected from the set {10, 20, 40, 50, 100, 200, 400, 500, 1000} ms. The utilization of each task in LO mode was generated using the UUnifast algorithm [4], such that the total LO-mode utilization across all tasks meets a given target, on average 80% for a core by default. We then calculated $C_i(LO) = u_i(LO)/T_i$. For HI-criticality tasks, $CFactor = C(HI)/C(LO)$ is then randomly selected within the range $[1, 2]$, and $C(HI)$ was calculated as $C(HI) = CFactor \cdot C(LO)$. The HI mode PFH was set to 1×10^{-9}, equivalent to the avionics safety standard DO-178C

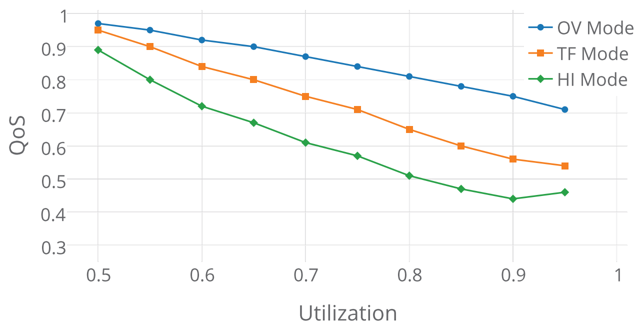


Fig. 3: Modes OV and TF achieve better QoS than HI for all utilizations (F not bounded).

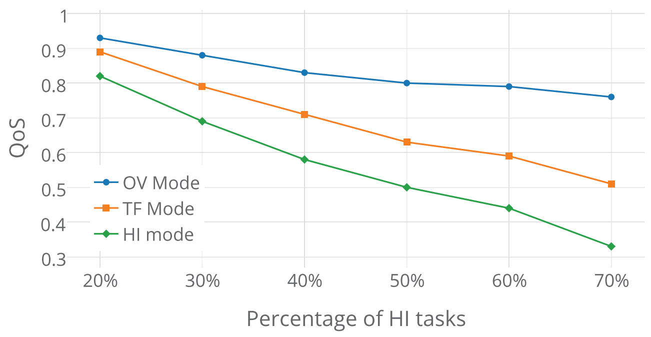


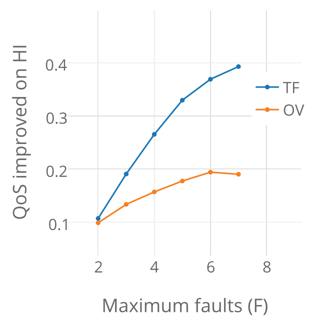
Fig. 4: Modes OV and TF achieve better QoS than HI for different percentages of HI tasks (F not bounded).

level-A requirements. The number of re-executions for each task in modes TF and HI was then derived using Equation (2). We assumed $\lambda_\pi = 1 \times 10^{-4}$. 1000 systems were generated for each combination of resource utilization, fraction of HI-criticality task utilization, and task count.

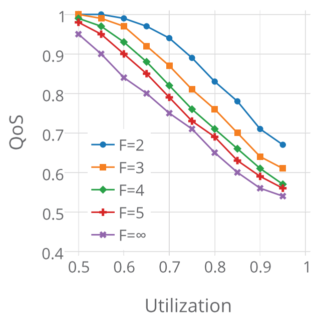
Figure 3 shows the performance of each mode as utilization is increased for a single-core architecture. 20 tasks were scheduled, half of which are HI. We observe that in general, LO task QoS is better in OV and TF than HI: on average OV and TF execute 42.9% and 20.2% more LO-criticality tasks than HI respectively. The benefit is relatively small at low processor utilization: CPU idle time is available to execute longer tasks (OV mode) and task re-executions (TF mode), or even a combination (HI mode). As utilization increases, the QoS in HI mode degrades as more LO-criticality tasks must be dropped to accommodate HI-criticality tasks.

Figure 4 shows the performance of each mode as the percentage of HI tasks is increased from 20% to 70% while utilization is maintained at 80%. All other parameters are similar to the ones in Figure 3. Modes OV and TF maintain better QoS than HI throughout the experiment. The benefit of these two modes becomes higher as the percentage of HI tasks in the system increases, hence requiring more CPU time for overruns and re-executions.

Figure 5a shows the average improvement for modes OV and TF compared to HI as a function of the maximum number of faults (F) considered. When fewer faults are considered, the HI mode performs better and the relative improvement is



(a)



(b)

Fig. 5: (a) Average improvement over all system utilizations for OV and TF modes compared to HI mode. (b) Performance of TF mode for different F.

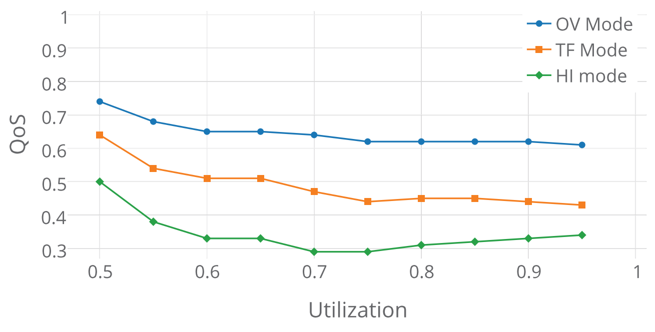


Fig. 6: Modes OV and TF achieve better QoS on multicores (F not bounded).

lower: less time is needed for task re-execution, freeing more resources for LO-criticality tasks in HI mode. Also, limiting pessimism on the analysis is an effective way to improve QoS.For example, in mode TF (Figure 5b), when $F = 2$, a 20.2% improvement in QoS is achieved by using the analysis in Section V-B compared to that of Section V-A.

Figure 6 illustrates the QoS for a multicore system (four lockstep cores). 40 tasks (20 HI, 20 LO) were scheduled on the four cores. Task allocation is performed in two steps: first HI tasks are sorted by decreasing utilization and assigned using a worst-fit policy. Then, LO tasks, sorted by decreasing utilization, were assigned using a best-fit policy. Previous results such as [1] indicate that this method provides good schedulability. As expected, OV and TF modes improve the QoS of LO tasks significantly over HI mode under lockstep, by 91.6% and 43.9% on average respectively. With more cores, there is more flexibility to efficiently utilize the processing resources in the new modes. This result shows that the model is beneficial to partitioned multicore architectures, by providing more significant QoS advantages for this increasingly popular platform in embedded systems.

VII. Conclusion

In this work, we proposed a novel four-mode model for mixed-criticality systems. The proposed model takes into account reliability requirements (specified as PFH) and criticality requirements (specified as additional conservative WCET estimates). The model also addresses designers' concerns about the lack of QoS guarantees for LO-criticality tasks. Significant increases in the QoS provided to LO tasks can be achieved with the new model. Schedulability analysis for the model is developed and improved by limiting the fault model pessimism.

Experimental results showed that the two new modes in the proposed model provide average improvements of 42.9% and 20.2% in QoS over the standard two-mode model. In multicore architectures, more substantial improvements (91.6% and 43.9%) can be obtained. These results indicate that improved QoS guarantees can be provided to LO tasks without adding hardware resources or affecting the guarantees to HI tasks.

Acknowledgments

This work has been partially supported by the Fonds de Recherche du Québec – Nature et Technologies (FQRNT) and the Natural Science and Engineering Research Council of Canada (NSERC).

References

[1] Z. Al-bayati, Q. Zhao, A. Youssef, H. Zeng, and Z. Gu, "Enhanced partitioned scheduling of Mixed-Criticality Systems on multicore platforms," *Asia and South Pacific Design Automation Conf.*, 630–635, 2015.

[2] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, M. Peri, and S. Pezzini, "Fault-tolerant platforms for automotive safety-critical applications," in *International conference on Compilers, architecture and synthesis for embedded systems*, 2003.

[3] S. Baruah, A. Burns, and R. Davis, "Response-time analysis for mixed criticality systems," in *32nd IEEE Real-Time Systems Symposium*, 2011.

[4] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, 30(1-2):129–154, 2005.

[5] A. Burns and S. Baruah, "Towards a more practical model for mixed criticality systems," *Workshop on Mixed Criticality Systems*, 2013.

[6] A. Burns and R. Davis, "Mixed criticality systems-a review," *Department of Computer Science, University of York, Tech. Rep*, 2015.

[7] P. Eles, V. Izosimov, P. Pop, and P. Zebo, "Synthesis of Fault-Tolerant Embedded Systems," *Design, Automation and Test in Europe*, 2008.

[8] M. Ebrahimi et al. "Comprehensive Analysis of Sequential and Combinational Soft Errors in an Embedded Processor," in *IEEE TCAD*, 34(10):1586-1599, Oct. 2015.

[9] T. Fleming and A. Burns, "Incorporating the notion of importance into mixed criticality systems," *Workshop on Mixed Criticality Sys.*, 2014.

[10] P. Huang, H. Yang, and L. Thiele, "On the scheduling of fault-tolerant mixed-criticality systems," in *Design Automation Conference*, 2014.

[11] S. Kang, H. Yang, S. Kim, I. Bacivarov, S. Ha, and L. Thiele, "Static Mapping of Mixed-Critical Applications for Fault-Tolerant MPSoCs," in *Design Automation Conference*, 2014.

[12] C. LaFrieda, E. Ipek, J. F. Martinez, and R. Manohar, "Utilizing dynamically coupled cores to form a resilient chip multiprocessor," in *IEEE/IFIP Int. Conf. on Dependable Systems and Networks*, 2007.

[13] J. Lin, A. M. Cheng, D. Steel, and M. Y.-C. Wu, "Scheduling mixed-criticality realtime tasks with fault tolerance," in *Workshop on Mixed Criticality Systems*, 2014.

[14] B. H. Meyer, B. H. Calhoun, J. Lach, and K. Skadron, "Cost-effective safety and fault localization using distributed temporal redundancy," *Compilers, architecture and synthesis for embedded systems*, 2011.

[15] R. Pathan, "Fault-tolerant and real-time scheduling for mixed-criticality systems," *Real-Time Systems*, 50(4):509–547, 2014.

[16] A. Thekkilakattil, R. Dobrin, and S. Punnekkat, "Mixed criticality scheduling in fault-tolerant distributed real-time systems," in *International Conference on Embedded Systems*, 2014.

[17] G. Upasani, X. Vera, and A. González, "Avoiding core's DUE & SDC via acoustic wave detectors and tailored error containment and recovery," in *Int. Symposium on Computer Architecuture*, 2014.

[18] E. Yip, M. M. Kuo, P. S. Roop, and D. Broman, "Relaxing the synchronous approach for mixed-criticality systems," in *IEEE Real-Time and Embedded Technology and Applications Symp.*, 2014.