Journal of Circuits, Systems, and Computers
Vol. 26, No. 1 (2017) 1750016 (17 pages)

DOI: 10.1142/S0218126617500165

World Scientific
www.worldscientific.com

Fault-Tolerant Task Scheduling for Mixed-Criticality Real-Time Systems*

Junlong Zhou, Min Yin, Zhifang Li, Kun Cao, Jianming Yan,
Tongquan Wei† and Mingsong Chen

Shanghai Key Laboratory of Trustworthy Computing,
East China Normal University,
Shanghai 200241, P. R. China
†*tqwei@cs.ecnu.edu.cn*

Xin Fu

Department of Electrical and Computer Engineering,
University of Houston,
Houston, TX 77204-4005, USA

Received 9 March 2016
Accepted 11 July 2016
Published 26 August 2016

Integration of safety-critical tasks with different certification requirements onto a common hardware platform has become a growing tendency in the design of real-time and embedded systems. In the past decade, great efforts have been made to develop techniques for handling uncertainties in task worst-case execution time, quality-of-service, and schedulability of mixed-criticality systems. However, few works take fault-tolerance as a design requirement. In this paper, we address the scheduling of fault-tolerant mixed-criticality systems to ensure the safety of tasks at different levels of criticalities in the presence of transient faults. We adopt task re-execution as the fault-tolerant technique. Extensive simulations were performed to validate the effectiveness of our algorithm. Simulation results show that our algorithm results in up to 15.8% and 94.4% improvement in system reliability and schedule feasibility as compared to existing techniques, which contributes to a more safe system.

Keywords: Fault-tolerant; mixed criticality; real-time systems; task scheduling.

1. Introduction

Unlike traditional embedded systems that almost have only one criticality level, many complex embedded systems nowadays are mixed-critical, where functionalities (or called tasks) of different importance co-exist and are provided with varying

*This paper was recommended by Regional Editor Zoran Stamenkovic.
†Corresponding author.

degrees of assurance. The safety-critical systems having two or more distinct criticality levels are named as mixed-criticality systems and the example of such systems can be found in avionics and automotive industries.[1–3] For example, the criticalities of drive control and music playing in a smart car system are obviously unequal, thus the two functionalities need to be provided with different degrees of assurance. Moreover, in such mixed-criticality systems, functionalities of higher importance should be assigned a higher criticality so that safety requirements for these functionalities could be ensured.

Considerable research efforts have been devoted to the design of mixed-criticality systems in the past decade. Vestal[4] initiated and formalized the mixed-criticality scheduling problem and proposed a preemptive fixed-priority algorithm for scheduling such systems, which was proved to be optimal in the scope of preemptive fixed-priority algorithms.[5] Baruah *et al.*[6] investigated the fixed-priority scheduling upon preemptive uniprocessors of mixed-criticality systems, presented a novel priority assignment scheme, and provided a sufficient response time analysis. Unlike Refs. 4–6, non-fixed-priority algorithms are also explored in Refs. 7 and 8 to schedule mixed-criticality systems in a certifiably correct manner. To solve the service abrupt problem for low-criticality (LO) tasks in conventional mixed-criticality scheduling algorithms, Su and Zhu[9] studied an elastic mixed-criticality task model that specifically allows low-criticality tasks to have variable periods. In addition, they also proposed an early-release earliest deadline first (EDF) scheduling algorithm for low-criticality tasks to improve their execution frequencies without violating timeliness of high-criticality (HI) tasks. However, all the above works did not take into account fault-tolerance.

For safety-critical embedded systems, the functionalities of these systems must be ensured under various stresses such as hardware/software (HW/SW) errors and power shortages.[10] In particular, fault-tolerance is imperative in the design of such systems to fight against potential failures for achieving high safety and reliability. There are many fault-tolerant techniques such as dual/triple modular redundancy,[11,12] re-execution,[13,14] and checkpointing with rollback[15,16] widely used in handling the occurrence of faults. Dual/triple modular redundancy is usually considered to achieve reliability against transient faults in multicore platforms, where multiple processing units execute identical copies for each task and their results are voted on to produce a single output. Re-execution is a technique that exploits the slack time on the processor to have recovery tasks, which are used to enhance the reliability of original tasks. Checkpointing with rollback-recovery is a popular fault-tolerant technique deployed in real-time embedded systems to maintain the system reliability. In checkpointing, the state of the system is saved on a stable storage at each checkpoint. When a transient fault occurs, the system rolls back to the most recent checkpoint and resumes the execution.

A few recent papers have focused on addressing the fault-tolerant task mapping and scheduling for mixed-criticality systems in the presence of transient faults.[10,17–19] In addition to explicitly modeling the safety requirements at different criticality

levels according to safety standards, Huang *et al.*[10] proposed a scheduling algorithm that jointly handles the safety and schedulability requirements for mixed-criticality systems. Kang *et al.*[17] presented a static optimal mapping with worst-case guarantees for mixed-critical applications running on fault-tolerant MPSoCs. A mixed-criticality scheduling with task dropping is also designed to ensure timeliness and worst-case response time of high-criticality applications. Pathan[18] modeled mixed-criticality systems from the perspective of fault-tolerance and proposed an approach to address temporal dependencies among real-time, fault-tolerance, and mixed-criticality constraints. Through the extension of classical HW/SW co-design paradigm, the application of fault-detection/tolerance techniques, and the exploitation of architecture features, a reliability-driven methodology is developed in Ref. 19, which can realize embedded systems with mixed-critical fault-management requirements. However, all the above works do not utilize idle time in a task schedule to improve the reliability and feasibility of the mixed-criticality system. Meanwhile, no mechanism was developed to enhance and ensure the priority of high-criticality tasks.

In this paper, we study the scheduling of mixed-criticality tasks executing on a uniprocessor platform in the presence of transient faults. The proposed fault-tolerant scheduling algorithm is developed based on earliest deadline first with virtual deadlines (EDF-VD).[20] Several techniques such as period transformation (PT), utilization of idle time (UIT), and re-execution are adopted in our algorithm to achieve a safe system ensuring high reliability and schedulability.

The remainder of the paper is organized as follows. Section 2 introduces the system models and problem definition. Section 3 shows the overall framework. Section 4 presents the proposed Slice-EDF-VD algorithm. Section 5 verifies the effectiveness of the proposed approach. The concluding remarks and discussion of future work are given in Sec. 6.

2. System Models and Problem Definition

We consider a task set Γ running on a uniprocessor system, where the processor is equipped with a fixed frequency. The processor can operate at two states, one is busy state and the other is idle state. If the processor executes tasks in a certain period of time, the processor is in the busy state and the period is called the busy time. Otherwise, the processor is in the idle state and the period when no tasks execute is called the idle time. For the sake of brevity, we focus on the mixed-criticality system with two different criticality levels in this work, which has a low criticality and a high criticality, and is referred to as dual-criticality system.[9,10,18,20,22] We leave the study of the mixed-criticality system with more criticality levels to our future work.

2.1. *Task model*

Consider a task set Γ that consists of n independent periodic real-time tasks $\{\tau_1, \tau_2, \ldots, \tau_i, \ldots, \tau_n\}$. The characteristics of every task τ_i $(1 \leq i \leq n)$ is described

by a quintuple $\{T_i, D_i, N_i, L_i, C_i\}$, where T_i is the period, D_i is the relative deadline, N_i is the maximum number of transient faults that can be tolerated during the task execution, $L_i \in \{\text{LO}, \text{HI}\}$ is the task criticality level, and C_i is the task worst-case execution time (WCET). In particular, $C_i(\text{LO})$ and $C_i(\text{HI})$ denote the WCETs of task τ_i in low- and high-criticality modes, respectively. The real-time constraint and the relations between low-criticality and high-criticality WCETs for every task can be formulated as

$$\forall \tau_i \in \Gamma : C_i(\text{LO}) \leq C_i(\text{HI}) \leq D_i \leq T_i \,. \tag{1}$$

The hyper-period of set Γ, denoted by H, is the least common multiple of periods $\{T_1, T_2, \ldots, T_n\}$. Every task τ_i could release a sequence (possibly infinite) of jobs (i.e., task instances $\tau_i^1, \tau_i^2, \tau_i^3, \ldots$) due to the property of periodicity. Let r_i^k denote the release time of the kth job, and it satisfies

$$\forall \tau_i \in \Gamma : r_i^{k+1} \geq r_i^k + T_i \,. \tag{2}$$

According to the criticality level of tasks, the set Γ can be partitioned into two subsets. Let $\text{LO}(\Gamma) = \{\tau_i \in \Gamma | L_i = \text{LO}\}$ represent the subset of low-criticality tasks in set Γ, and $\text{HI}(\Gamma) = \{\tau_i \in \Gamma | L_i = \text{HI}\}$ represent the subset of high-criticality tasks in set Γ. Then the processor utilization of executing task set Γ can be calculated as

$$U = \sum_{\tau_i \in \text{LO}(\Gamma)} \frac{C_i(\text{LO})}{T_i} + \sum_{\tau_i \in \text{HI}(\Gamma)} \frac{C_i(\text{HI})}{T_i} \,. \tag{3}$$

As reported in Ref. 22, tasks with different criticality levels are associated with different reliability requirements. That is, the reliability of high-criticality tasks is in general required to be above a threshold for safety guarantee and also higher than that of low-criticality tasks. To this end, we assume the system utilizes fault-tolerant technique to achieve a high reliability for high-criticality tasks $\tau_i \in \text{HI}(\Gamma)$, whereas executes low-criticality tasks $\tau_i \in \text{LO}(\Gamma)$ in a best-effort manner.

2.2. *Fault model*

Transient fault is a type of failure that appears for a short time and then disappears without damage to the device, and is caused by electromagnetic interference or cosmic radiation. It is indispensable for many safety-related embedded systems to have the capacity of providing a reliable execution in the presence of transient faults. Transient faults are in general modeled using an exponential distribution with an average arrival rate λ, which represents the expected number of failures occurring per second.[23] The reliability of a task is defined as the probability of its successful execution with no occurrence of transient faults, and can be determined by using the exponential failure law. Specifically, given the fault arrival rate λ, the reliability R_i of

task τ_i is expressed as[23]

$$R_i = e^{-\lambda C_i}\,, \tag{4}$$

where C_i is the worst-case execution time of the task.

Re-execution has been widely used in improving system reliability due to transient faults.[17] It assumes that the fault is locally detected at the end of task execution and it re-executes the task when a fault is detected. It utilizes a vote device to compare the execution results and selects the correct one as the output. We adopt re-execution in this paper to enhance the system reliability. Let Υ_i denote the number of times that task τ_i executes (including re-execution) to tolerate N_i faults, and it is given by

$$\Upsilon_i = 2N_i + 1\,. \tag{5}$$

We consider systems that use re-execution to tolerate up to one transient fault since single-fault-tolerance is a common assumption.[24] We then have $\Upsilon_i = 1$ ($N_i = 0$) if $\tau_i \in \mathrm{LO}(\Gamma)$ and $\Upsilon_i = 3$ ($N_i = 1$) if $\tau_i \in \mathrm{HI}(\Gamma)$ since high-criticality tasks are required to deliver a high reliability while low-criticality tasks are not. Therefore, based on (4), the reliability of task τ_i with re-execution technique is

$$R(\tau_i) = \begin{cases} e^{-\lambda C_i(\mathrm{LO})} & \text{if} \quad \tau_i \in \mathrm{LO}(\Gamma)\,, \\ 1 - (1 - e^{-\lambda C_i(\mathrm{HI})})^3 & \text{if} \quad \tau_i \in \mathrm{HI}(\Gamma)\,. \end{cases} \tag{6}$$

2.3. *Problem definition*

Fault-tolerant mixed-criticality scheduling problem: Given a periodic real-time task set Γ running on a dual-criticality system, the fault profile N_i for all tasks in set Γ, and an average error rate λ, find a fault-tolerant scheduling algorithm, which can enhance reliability and schedulability/feasibility for ensuring system safety.

3. Framework

In this section, we first propose a metric to quantify the safety of the concerned dual-criticality system and show the necessity of task dropping to improve the schedulability. We then introduce a scheduling algorithm and two techniques that are adopted in our scheme to help achieve a safe system.

3.1. *Safety metric*

For a safe dual-criticality system, the reliability of high-criticality tasks should be maintained at a high level and both of low-criticality and high-criticality tasks should complete their execution before their deadlines. Taking this into account, we

first propose a metric to quantify the system safety during a hyper-period H, which is the product of the probability that high-criticality tasks execute successfully in the absence of transient faults and the ratio of the number ($\mathcal{M}$) of task instances that finish their execution in time to the total number ($\sum_{\tau_i \in \Gamma} \frac{H}{T_i}$) of task instances. It is formulated as

$$S(\Gamma, H) = \left[\prod_{\tau_i \in \mathrm{HI}(\Gamma)} (R(\tau_i))^{\frac{H}{T_i}} \right] \cdot \frac{\mathcal{M}}{\sum_{\tau_i \in \Gamma} \frac{H}{T_i}} \,. \tag{7}$$

Obviously, the system safety given in (7) can be improved by increasing the reliability of high-criticality tasks and the real-time feasibility of task set. However, using re-execution to improve task reliability would lead to a higher processor utilization, which may result in an increased number of tasks missing their deadlines. Thus, in order to achieve a high system safety, the trade-off between reliability and real-time feasibility needs to be balanced.

3.2. *Need for task dropping*

According to (3) and (5), the utilization U of executing the task set Γ with re-execution can be written as

$$\begin{aligned} U &= \sum_{\tau_i \in \mathrm{LO}(\Gamma)} \frac{C_i(\mathrm{LO})\Upsilon_i}{T_i} + \sum_{\tau_i \in \mathrm{HI}(\Gamma)} \frac{C_i(\mathrm{HI})\Upsilon_i}{T_i} \\ &= \sum_{\tau_i \in \mathrm{LO}(\Gamma)} \frac{C_i(\mathrm{LO})(2N_i + 1)}{T_i} + \sum_{\tau_i \in \mathrm{HI}(\Gamma)} \frac{C_i(\mathrm{HI})(2N_i + 1)}{T_i} \,. \end{aligned} \tag{8}$$

The utilization U needs to satisfy the constraint

$$U \leq 1 \,, \tag{9}$$

if the task set Γ is schedulable on the processor.

However, with the increasing time overhead caused by re-execution, the task set may become unschedulable. Table 1 gives an example that task set Γ consisting of two low-criticality tasks and one high-criticality task cannot be scheduled by any algorithms since $U = 1.28 > 1$. In order to guarantee the schedulability of task set, the utilization should be below the upper bound, as shown in (9). Two approaches are commonly used to improve the schedulability of task set by reducing the utilization. One approach is to choose a higher frequency to execute tasks. However, this approach is not suitable for our model that assumes the processor frequency is fixed, and also more energy would be consumed when executing tasks at a higher frequency. Dropping less critical tasks is the other useful approach to reduce the utilization and hence improve the schedulability of task set. As demonstrated in Table 2, the utilization U is lowered to 0.95 and the set Γ becomes schedulable after dropping a low-criticality task τ_1.

Table 1. $U = \frac{2}{6} + \frac{2}{10} + 3 \times \frac{3}{12} = 1.28 > 1$. The task set $\Gamma = \{\tau_1, \tau_2, \tau_3\}$ cannot be scheduled.

τ_i	C_i	T_i	D_i	L_i	N_i	Υ_i
τ_1	2	6	6	LO	0	1
τ_2	2	10	10	LO	0	1
τ_3	3	12	12	HI	1	3

Table 2. $U = \frac{2}{10} + 3 \times \frac{3}{12} = 0.95 < 1$. The task set $\Gamma = \{\tau_2, \tau_3\}$ can be scheduled.

τ_i	C_i	T_i	D_i	L_i	N_i	Υ_i
τ_2	2	10	10	LO	0	1
τ_3	3	12	12	HI	1	3

The aforementioned example shown in Tables 1 and 2 motivates the need for dropping less critical tasks from the schedulability point of view. Therefore, the approach of task dropping is adopted in this paper to guarantee the utilization constraint given in (9).

3.3. *Application of EDF-VD*

As indicated in (7), the safety of a dual-criticality system depends on the reliability of high-criticality tasks and the real-time feasibility of the whole task set. In addition, some high-criticality tasks may miss their deadlines due to re-execution, which could result in a decreased real-time feasibility and hence lower the system safety. Therefore, to achieve a safe mixed-criticality system, high-criticality tasks should be given a higher priority to execute, no matter whether the system is viewed from the standpoint of reliability or real-time feasibility. As far as we know, the scheduling algorithm EDF-VD presented in Ref. 20 is an effective method that can ensure a higher priority for a task with higher criticality, and can also achieve a better schedulability as compared to traditional scheduling algorithm (e.g., EDF[21]). Thus, we adopt this method in our task scheduling scheme. The EDF-VD operates as follows. It first presents a concept of virtual deadline and calculates the virtual deadlines for all tasks. It then utilizes these virtual deadlines to determine task scheduling priority according to the EDF policy. We briefly summarize the calculation of virtual deadline below, and suggest the readers to refer to Ref. 20 for further details.

The virtual deadline of job τ_i^k is represented by VD_i^k and formulated as

$$\mathrm{VD}_i^k = r_i^k + D_i - \Delta_i \,, \tag{10}$$

where r_i^k is the release time, D_i is the relative deadline, and Δ_i is the time offset between real deadline and virtual deadline. Assuming two jobs τ_i^k and τ_j^k are released

at the same time (i.e., $r_i^k = r_j^k$) and have the same relative deadline (i.e., $D_i = D_j$), then the two jobs should satisfy the relation

$$L_i \leq L_j \Rightarrow \Delta_i \leq \Delta_j \Rightarrow \mathrm{VD}_i^k \geq \mathrm{VD}_j^k, \tag{11}$$

which indicates the task with lower criticality has a larger virtual deadline. Then, according to the principle of earliest virtual deadline first, tasks with higher criticality would be assigned a higher priority to execute.

To be specific, in the concerned dual-criticality system, the time offset Δ_i of a high-criticality task is set to C_i while the Δ_i of a low-criticality task is set to 0, that is,

$$\mathrm{VD}_i^k = \begin{cases} r_i^k + D_i - C_i & \text{if } L_i = \mathrm{HI}, \\ r_i^k + D_i & \text{if } L_i = \mathrm{LO}. \end{cases} \tag{12}$$

In this way, high-criticality tasks can always have a higher priority than low-criticality tasks. Furthermore, when a high-criticality task $\tau_i \in \mathrm{HI}(\Gamma)$ misses its virtual deadline, all the low-criticality tasks in $\mathrm{LO}(\Gamma)$ would be dropped to generate extra available time for re-executing task τ_i and the next high-criticality tasks violating the virtual deadline constraint.

3.4. *Two techniques to refine the scheduling feasibility*

As introduced above, the EDF-VD can elevate the priority of high-criticality tasks to generate a feasible fault-tolerant mixed-criticality task schedule. In addition, two techniques can be used to refine the scheduling of fault-tolerant mixed-criticality tasks, as described below.

Period transformation: PT is an applicable technique that has been widely used in the scheduling of mixed-criticality tasks, and it can improve the multi-criticality schedulability by ensuring tasks with higher criticalities have higher scheduling priorities.[25] PT splits a task τ_i with period T_i, relative deadline D_i, and WCET C_i into X_i parts, such that the task then has smaller period $\frac{T_i}{X_i}$, relative deadline $\frac{D_i}{X_i}$, and WCET $\frac{C_i}{X_i}$. For example, a task τ_i with parameters $T_i = D_i = 10, C_i = 4$ becomes the task with parameters $T_i = D_i = 5, C_i = 2$ through one operation of PT. Clearly, the relative deadline (and hence the virtual deadline) is decreased during the operation of PT. If all the high-criticality tasks are transformed using PT, their transformed deadlines are shorter than those of low-criticality tasks. Then, all the high-criticality tasks will have higher priorities as compared to low-criticality tasks according to the principle of EDF. However, extra overheads from the increased number of context switches are inevitable by introducing PT.

Utilization of idle time: Note that the processor usually has some idle time, which is dispersedly located between the task release time and start time of task execution, or between the end time of task execution and task deadline. As aforementioned,

high-criticality tasks demand more execution time since they use re-execution to enhance the system reliability. Thus, these idle times can be assigned to high-criticality tasks for satisfying their requirements. However, such time slots are typically quite small so that they are difficult to be directly utilized for executing high-criticality tasks.

Problem conversion: Based on the above analysis, if we apply PT and UIT in the scheme EDF-VD, both the reliability and schedulability of task set Γ could be improved. In addition, the limitations of the two techniques need to be overcome when we adopt them in EDF-VD. Thus, the key of our problem defined in Sec. 2.3 becomes

- How to apply PT to preprocess the task set Γ to ensure the priority of high-criticality tasks without incurring large context switch overhead?
- How to make UIT feasible and apply it for executing high-criticality tasks?

4. The Proposed Slice-EDF-VD Algorithm

To solve the above two issues, we propose an approach called Slice-EDF-VD that is developed based on the EDF-VD algorithm and integrates itself with PT and UIT.

4.1. *Preprocess the task set*

The objective of our fault-tolerant mixed-criticality task scheduling is to tolerate N_i transient faults for every high-criticality task τ_i in subset $\mathrm{HI}(\Gamma)$. When we use re-execution to enhance the reliability of high-criticality tasks (e.g., τ_i) in the presence of transient faults, the task execution time becomes Υ_i times of it. Notice that all the re-executions of a high-criticality task are independent and exactly the same as the original task, which motivates us to treat the task and its redundancies as Υ_i identically independent and isolated task instances (or called task slices). This can be achieved by using a modified-PT that does not split the code and hence would not increase the complexity of system design due to homogeny of the task and its redundancies.

The modified-PT operates as follows. It converts the high-criticality task with its redundancies into multiple task instances (slices) with updated execution times and virtual deadlines. Specifically, the high-criticality task τ_i with its redundancies is transformed into Υ_i new tasks $\tau_{i,1}, \tau_{i,2}, \ldots, \tau_{i,\Upsilon_i}$, where every new task $\tau_{i,j}$ $(1 \leq j \leq \Upsilon_i)$ has the execution time of C_i and the virtual deadline of $r_i + \max\{0, \frac{D_i}{\Upsilon_i} \times j - C_i\}$. For instance, consider a set Γ comprised of a low-criticality task τ_1 $(D_1 = T_1 = 4, L_1 = \mathrm{LO}, C_1(\mathrm{LO}) = C_i = 2)$ and a high-criticality task τ_2 $(D_2 = T_2 = 15, L_2 = \mathrm{HI}, C_2(\mathrm{HI}) = 3C_i = 6)$. Using the modified-PT method, the task set can be transformed into a set comprised of four tasks $\tau_{1,1}, \tau_{2,1}, \tau_{2,2}$ and $\tau_{2,3}$, which is listed in Table 3. As shown in the table, through the transformation of high-criticality task τ_2, the new generated tasks $\tau_{2,1}, \tau_{2,2}$ and $\tau_{2,3}$ are shorter than the

Table 3. The transformed set of four tasks $\tau_{1,1}, \tau_{2,1}, \tau_{2,2}$ and $\tau_{2,3}$. (Note that $\tau_{1,1}$ is the same task as τ_1 since modified-PT is only applied to high-criticality task.)

τ_i	C_i(LO) or C_i(HI)	T_i	D_i	VD_i	L_i	Υ_i
$\tau_{1,1}$	2	4	4	4	LO	1
$\tau_{2,1}$	2	15	15	3	HI	3
$\tau_{2,2}$	2	15	15	8	HI	3
$\tau_{2,3}$	2	15	15	13	HI	3

original task τ_2 such that the dispersed small idle time slots can be adequately utilized.

4.2. *Task scheduling algorithm*

We now present our proposed fault-tolerant task scheduling algorithm. The input to our algorithm is a dual-criticality task set $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ running on a uniprocessor system. The algorithm works as follows. It first makes a preprocess to the high-criticality tasks in the task set Γ using the modified-PT technique (line 1), as described in Sec. 4.1. The resultant task set is comprised of $\sum_{i=1}^{n} \Upsilon_i$ tasks and can be represented by

$$\Gamma' = \{\tau_{1,1}, \tau_{1,2}, \ldots, \tau_{1,\Upsilon_1}, \ldots, \tau_{n,1}, \tau_{n,2}, \ldots, \tau_{n,\Upsilon_n}\}.$$

It then initializes the ready queue Q_{ready} to null (line 2). As introduced in Sec. 2.1, the system will release its job $\tau_{i,j}^k$ every T_i time units for each task $\tau_{i,j}$ in the set Γ'. In other words, new jobs would keep rolling in when the system is turned on. Therefore, an always-true loop (lines 3–21) is adopted in the algorithm to handle this case. In each round of iteration, the algorithm pushes all the released jobs into the queue Q_{ready}, then dequeues the ready jobs for execution from Q_{ready} according to the policy of EDF-VD if the processor is idle and Q_{ready} is not empty (lines 4–6).

For high-criticality tasks, the algorithm utilizes a hash-based mechanism *fingerprinting*[26] to detect transient faults, which is timely and bandwidth-efficient, and has an arbitrary good fault-detection coverage. Fault detection using *fingerprinting* operates as follows.[27] Given the input in the form of a program (algorithm) and some data, the fingerprint is determined by the instruction and data sequence imposed by the program. The generated fingerprint will always be the same as long as program and data are the same; otherwise, there must be some faults. This motivates us to detect whether a fault has happened or not by comparing the fingerprints of high-criticality tasks (lines 7–16). Thus, the algorithm memorizes the fingerprints of high-criticality tasks during the task execution. It employs a voting device to compare the fingerprints of $\tau_{i,1}^k, \tau_{i,2}^k, \ldots, \tau_{i,\Upsilon_i-1}^k$ for high-criticality task instance τ_i^k. If their fingerprints are the same which indicates task executions are correct, the flag variable

Algorithm 1. Pseudo-code of Slice-EDF-VD

Input: A dual-criticality task set $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$

1: $\Gamma' \leftarrow$ preprocess Γ using the modified-PT;
2: $Q_{\text{ready}} \leftarrow \phi$;
3: **while True do**
4: push the released jobs $\tau_{i,j}^k$ from Γ' into Q_{ready};
5: **if** $State = idle$ and $Q_{\text{ready}} \neq \phi$ **then**
6: dequeue the ready job $\tau_{i,j}^k$ from Q_{ready} according to the EDF-VD;
7: **if** $L_i = \text{HI}$ **then**
8: **for** $j = 1$ to $\Upsilon_i - 1$ **do**
9: execute $\tau_{i,j}^k$ and memorize fingerprint;
10: **end for**
11: $\text{Result}_i^k \leftarrow$ vote $\tau_{i,1}^k, \tau_{i,2}^k, \ldots, \tau_{i,\Upsilon_i-1}^k$;
12: **if** $\text{Result}_i^k = Correct$ **then**
13: drop τ_{i,Υ_i}^k from Q_{ready}; {*no error, thus no re-execution*}
14: **else**
15: drop all low-criticality tasks from Q_{ready} and execute τ_{i,Υ_i}^k; {*re-execution when error exists*}
16: **end if**
17: **else**
18: execute $\tau_{i,j}^k$;
19: **end if**
20: **end if**
21: **end while**

Result_i^k of τ_i^k is set to *Correct*. Accordingly, τ_{i,Υ_i}^k is dropped from Q_{ready} since there is no need of re-execution. Otherwise, tasks with low criticality are dropped for ensuring the re-execution of high-criticality tasks. Without the fault-tolerance requirement of low-criticality tasks, *fingerprinting* is not applied to low-criticality tasks (lines 17–19). The pseudo-code of the proposed algorithm, named Slice-EDF-VD, is described in Algorithm 1.

5. Evaluation

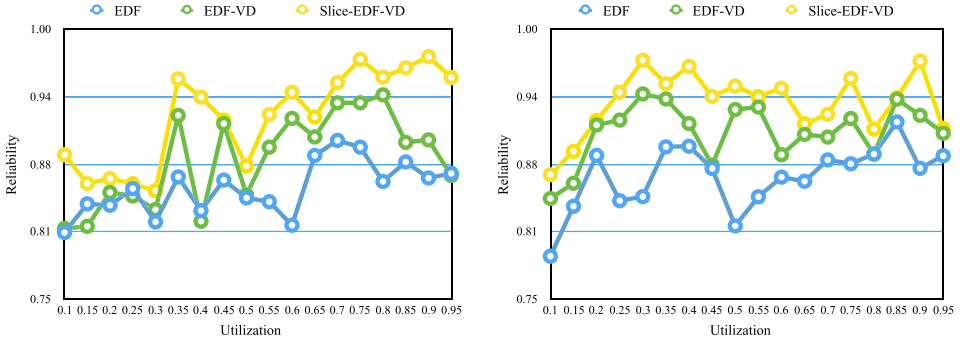
According to the safety metric that is defined in (7) which shows that the safety of a dual-criticality system depends on the system reliability and real-time feasibility, we first carry out two sets of simulation experiments to validate the proposed scheduling algorithm Slice-EDF-VD with respect to the two aspects, respectively. The proposed algorithm Slice-EDF-VD is compared with the benchmarking algorithms EDF[21] and EDF-VD[20] in the simulation. The two benchmarking algorithms adopted in the comparison are described as follows. EDF is a scheduling algorithm that prioritizes

active tasks according to their deadlines, that is, the earlier the deadline, the higher the priority.[21] EDF-VD is extended from EDF and designed for the scheduling of mixed-criticality task systems.[20] With the application of virtual deadline, it can ensure a higher priority for a task with higher criticality and also achieve a better schedulability as compared to traditional scheduling algorithms. After the two sets of simulation experiments, we further investigate the CPU time overhead and scalability of the proposed algorithm Slice-EDF-VD.

All the algorithms were implemented in Python 2.7.8 and simulations were performed on a machine with Intel Quad-Core 2.1 GHz processor and 6 GB memory. Task sets used in the simulation are constructed by a random mixed-criticality task generation algorithm presented in Ref. 28, which determines the number of tasks in a set, the criticality levels, release times, WCET, periods, and deadlines of tasks. The average fault arrival rate λ is set to be 1×10^{-5}. In our simulations, the value of Υ_i for a low-criticality task $\tau_i \in \mathrm{LO}(\Gamma)$ is set to 1, and the value of Υ_i for a high-criticality task $\tau_i \in \mathrm{HI}(\Gamma)$ can be set to 3 and 5. In other words, $N_i = 1$ and $N_i = 2$ are utilized for fault-tolerance in the evaluation. $N_i = 1$ has been widely utilized in the previous works since single-fault-tolerance is a common assumption.[24] This is because that a large N_i could lead to a high task execution time and system utilization, which may violate the constraints of system utilization and task deadline. However, in order to do more investigations of our method and the benchmarking methods, and simultaneously have a relatively high system schedulability, we utilize $N_i = 1$ and $N_i = 2$ in the evaluation.

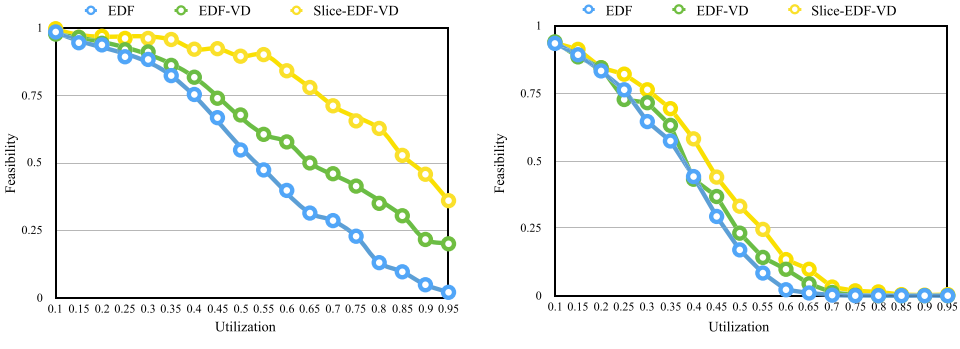
Figure 1 shows the variations of reliability of the three algorithms EDF,[21] EDF-VD,[20] and Slice-EDF-VD under varying processor utilizations. It has been demonstrated in the figure that our Slice-EDF-VD algorithm has a higher reliability as compared to benchmarking algorithms EDF[21] and EDF-VD.[20] More specifically, in the case of $\Upsilon_i = 3$ for high-criticality tasks, the reliability achieved by Slice-EDF-VD is 7.8% and 4.6% higher than those of EDF[21] and EDF-VD[20] on average, respectively. In the case of $\Upsilon_i = 5$ for high-criticality tasks, the reliability achieved by Slice-EDF-VD is 8.0% and 2.99% higher than those of EDF[21] and EDF-VD[20] on average, respectively. For the two cases, the reliability improvements achieved by Slice-EDF-VD over EDF[21] and EDF-VD[20] can be up to 15.6% and 15.8%, respectively. As shown in the figure, the reliabilities of the three algorithms in the case of $\Upsilon_i = 5$ are higher than those in the case of $\Upsilon_i = 3$, which is due to the increasing re-executions.

Figure 2 presents the real-time feasibility those of the three algorithms EDF,[21] EDF-VD,[20] and Slice-EDF-VD under varying processor utilizations. As can be seen from the figure, our Slice-EDF-VD algorithm has a higher feasibility as compared to benchmarking algorithms EDF[21] and EDF-VD.[20] To be specific, in the case of $\Upsilon_i = 3$ for high-criticality tasks, the feasibility achieved by Slice-EDF-VD is 41.7% and 24.2% higher than those of EDF[21] and EDF-VD[20] on average, respectively. In the case of $\Upsilon_i = 5$ for high-criticality tasks, the feasibility achieved by Slice-EDF-VD is



(a) $\Upsilon_i = 1$ if $\tau_i \in \text{LO}(\Gamma)$ and $\Upsilon_i = 3$ if $\tau_i \in \text{HI}(\Gamma)$. (b) $\Upsilon_i = 1$ if $\tau_i \in \text{LO}(\Gamma)$ and $\Upsilon_i = 5$ if $\tau_i \in \text{HI}(\Gamma)$.

Fig. 1. The variations of reliability of the three algorithms EDF,[21] EDF-VD,[20] and Slice-EDF-VD under varying processor utilizations.

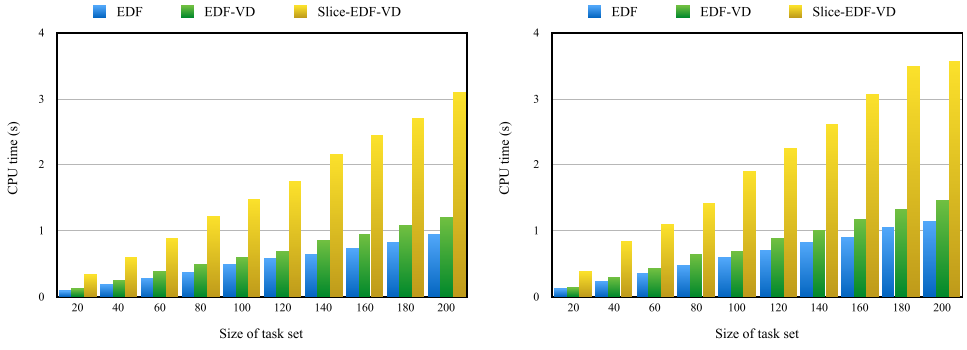


(a) $\Upsilon_i = 1$ if $\tau_i \in \text{LO}(\Gamma)$ and $\Upsilon_i = 3$ if $\tau_i \in \text{HI}(\Gamma)$. (b) $\Upsilon_i = 1$ if $\tau_i \in \text{LO}(\Gamma)$ and $\Upsilon_i = 5$ if $\tau_i \in \text{HI}(\Gamma)$.

Fig. 2. The real-time feasibility variations of the three algorithms EDF,[21] EDF-VD,[20] and Slice-EDF-VD under varying processor utilizations.

57.7% and 43.2% higher than those of EDF[21] and EDF-VD[20] on average, respectively. For the two cases, the feasibility improvements achieved by Slice-EDF-VD over EDF[21] and EDF-VD[20] can be up to 94.4% and 88.9%, respectively. It also has been demonstrated in the figure that the feasibilities of the three algorithms in the case of $\Upsilon_i = 5$ are lower than those in the case of $\Upsilon_i = 3$, which is due to the heavier workload caused by the increasing re-executions.

To have an overall evaluation of our algorithm, we compare the CPU times consumed by the three algorithms EDF,[21] EDF-VD,[20] and our proposed Slice-EDF-VD when executing task sets with varying sizes. Ten sets comprised of 20, 40, 60, 80, 100, 120, 140, 160, 180, and 200 tasks are utilized in the evaluation. The results given in Fig. 3 show that Slice-EDF-VD consumes a longer CPU time as compared to EDF[21] and EDF-VD[20] due to the adoption of *fingerprinting*. For example, in the case

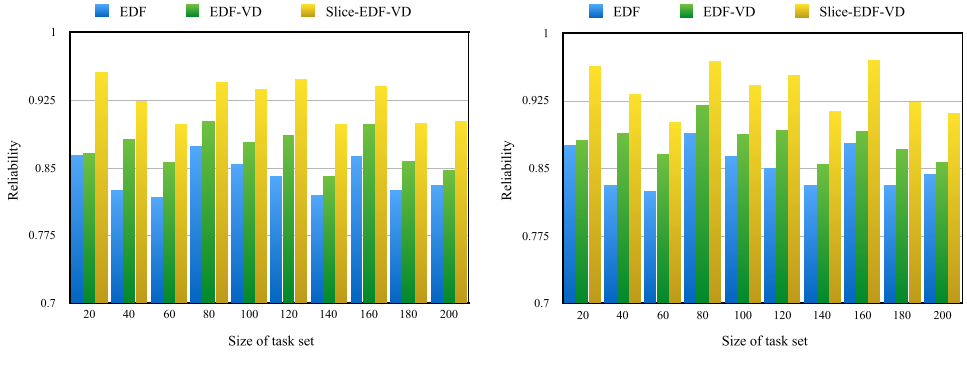


(a) $\Upsilon_i = 1$ if $\tau_i \in \mathrm{LO}(\Gamma)$ and $\Upsilon_i = 3$ if $\tau_i \in \mathrm{HI}(\Gamma)$. (b) $\Upsilon_i = 1$ if $\tau_i \in \mathrm{LO}(\Gamma)$ and $\Upsilon_i = 5$ if $\tau_i \in \mathrm{HI}(\Gamma)$.

Fig. 3. The CPU times consumed by the three algorithms EDF,[21] EDF-VD,[20] and Slice-EDF-VD when executing task sets with varying sizes.

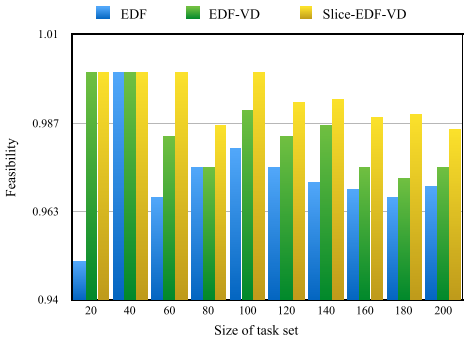
of $\Upsilon_i = 3$ for high-criticality tasks, the CPU times consumed by EDF,[21] EDF-VD,[20] and Slice-EDF-VD are 0.5106, 0.6588, and 1.6647 s, respectively. In the case of $\Upsilon_i = 5$ for high-criticality tasks, the CPU times consumed by EDF,[21] EDF-VD,[20] and Slice-EDF-VD are 0.6374, 0.8031, and 2.0619 s, respectively. As demonstrated in the results, the CPU time overhead of Slice-EDF-VD is higher than those of EDF[21] and EDF-VD,[20] but it is still acceptable due to its little magnitude.

Furthermore, in order to investigate the scalability of our algorithm in terms of increasing the number of task executions, we compare the reliability and real-time feasibility achieved by Slice-EDF-VD with those of EDF[21] and EDF-VD[20] when executing task sets with varying sizes. The sizes of ten task sets take the values of 20, 40, 60, 80, 100, 120, 140, 160, 180, and 200, respectively. The results given in Fig. 4 clearly show that the proposed Slice-EDF-VD outperforms EDF[21] and EDF-VD[20]

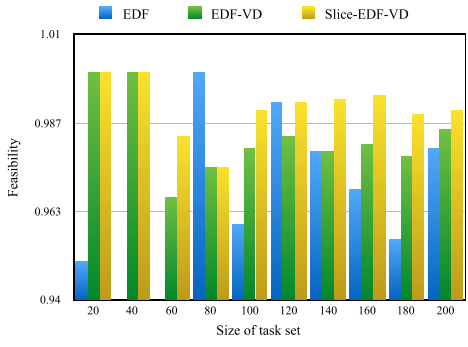


(a) $\Upsilon_i = 1$ if $\tau_i \in \mathrm{LO}(\Gamma)$ and $\Upsilon_i = 3$ if $\tau_i \in \mathrm{HI}(\Gamma)$. (b) $\Upsilon_i = 1$ if $\tau_i \in \mathrm{LO}(\Gamma)$ and $\Upsilon_i = 5$ if $\tau_i \in \mathrm{HI}(\Gamma)$.

Fig. 4. The reliabilities and real-time feasibilities of the three algorithms EDF,[21] EDF-VD,[20] and Slice-EDF-VD when executing task sets with varying sizes.



(c) $\Upsilon_i = 1$ if $\tau_i \in \mathrm{LO}(\Gamma)$ and $\Upsilon_i = 3$ if $\tau_i \in \mathrm{HI}(\Gamma)$.



(d) $\Upsilon_i = 1$ if $\tau_i \in \mathrm{LO}(\Gamma)$ and $\Upsilon_i = 5$ if $\tau_i \in \mathrm{HI}(\Gamma)$.

Fig. 4. (*Continued*)

with respect to reliability and real-time feasibility, regardless of the size of task set. In addition, this conclusion can be drawn in the case of both $\Upsilon_i = 3$ and $\Upsilon_i = 5$ for high-criticality tasks, as illustrated in Fig. 4.

6. Conclusion and Future Work

In this paper, we propose a fault-tolerant algorithm for scheduling the dual-criticality tasks on a uniprocessor platform in the presence of transient faults. Our proposed Slice-EDF-VD algorithm adopts re-execution to improve the reliability and utilizes PT and UIT to enhance schedule feasibility. Through the efforts made in the two aspects, the safety of the system can be upgraded. Accordingly, we have conducted two sets of simulation experiments to validate our algorithm Slice-EDF-VD. The algorithm was shown to improve the reliability by up to 15.8% and the feasibility by up to 94.4% as compared to benchmarking algorithms. In the future, we plan on extending our work to consider multicore platforms and to consider tasks with more criticality levels.

Acknowledgments

This research was partially funded by the Shanghai Municipal NSF under the Grant No. 16ZR1409000, NSFC under the Grant No. 91418203, and East China Normal University Outstanding Doctoral Dissertation Cultivation Plan of Action under the Grant No. PY2015047.

References

1. M. Volp, M. Roitzsch and H. Hartig, Towards an interpretation of mixed criticality for optimistic scheduling, *Proc. Int. Symp. Real-Time and Embedded Technology and Application* (2015), pp. 15–16.

2. A. Burns and R. Davis, Mixed criticality systems — A review, Technical Report, Department of Computer Science, University of York (2013).
3. P. Huang, P. Kumar, G. Giannopoulou and L. Thiele, Energy efficient DVFS scheduling for mixed-criticality systems, *Proc. Int. Conf. Embedded Software* (2014), pp. 12–17.
4. S. Vestal, Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance, *Proc. Int. Symp. Real-Time Systems* (2007), 239–243.
5. F. Dorin, P. Richard, M. Richard and J. Goossens, Schedulability and sensitivity analysis of multiple criticality tasks with fixed-priorities, *Real-Time Syst.* **46** (2010) 305–331.
6. S. Baruah, A. Burns and R. Davis, Response-time analysis for mixed criticality systems, *Proc. Int. Symp. Real-Time Systems* (2011), pp. 34–43.
7. S. Braruah and S. Vestal, Schedulability analysis of sporadic tasks with multiple criticality specifications, *Proc. Int. Euromicro Conf. Real-Time Systems* (2008), pp. 147–155.
8. H. Li and S. Baruah, An algorithm for scheduling certifiable mixed-criticality sporadic task systems, *Proc. Int. Symp. Real-Time Systems* (2010), pp. 183–192.
9. H. Su and D. Zhu, An elastic mixed-criticality task model and its scheduling algorithm, *Proc. Int. Conf. Design, Automation and Test in Europe* (2013), pp. 147–152.
10. P. Huang, H. Yang and L. Thiele, On the scheduling of fault-tolerant mixed-criticality systems, *Proc. Int. Conf. Design Automation* (2014), pp. 1–6.
11. E. Elnozahy, R. Melhem and D. Mosse, Energy-efficient duplex and TMR real-time systems, *Proc. Int. Symp. Real-Time Systems* (2002), pp. 256–266.
12. M. Salehi, A. Ejlali and B. Al-Hashimi, Two-phase low-energy N-modular redundancy for hard real-time multi-core systems, *IEEE Trans. Parallel Distrib. Syst.* **27** (2016) 1497–1510.
13. J. Zhou, X. S. Hu, Y. Ma and T. Wei, Balancing lifetime and soft-error reliability to improve system availability, *Proc. Int. Conf. Asia and South Pacific Design Automation* (2016), pp. 685–690.
14. B. Zhao, H. Aydin and D. Zhu, Shared recovery for energy efficiency and reliability enhancements in real-time applications with precedence constraints, *ACM Trans. Des. Autom. Electron. Syst.* **18** (2013) 2.
15. Y. Zhang and K. Chakrabarty, A unified approach for fault tolerance and dynamic power management in fixed-priority real-time embedded systems, *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **25** (2006) 111–125.
16. J. Zhou and T. Wei, Stochastic thermal-aware real-time task scheduling with considerations of soft errors, *J. Syst. Softw.* **102** (2015) 123–133.
17. S. Kang, H. Yang, S. Kim, I. Bacivarov, S. Ha and L. Thiele, Static mapping of mixed-critical applications for fault-tolerant MPSoCs, *Proc. Int. Conf. Design Automation* (2014), pp. 1–6.
18. R. Pathan, Fault-tolerant and real-time scheduling for mixed-criticality systems, *Real-Time Syst.* **50** (2014) 509–547.
19. C. Bolchini and A. Miele, Reliability-driven system-level synthesis for mixed-critical embedded systems, *IEEE Trans. Comput.* **62** (2013) 2489–2502.
20. S. Braruah *et al.*, The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems, *Proc. Int. Euromicro Conf. Real-Time Systems* (2012), pp. 145–154.
21. O. Zapata and P. Alvarez, EDF and RM multiprocessor scheduling algorithms: Survey and performance evaluation, Report No. CINVESTAV-CS-RTG-02, Seccion de Computacion Av. IPN (2005).

22. P. Ekberg and Y. Wang, Bounding and shaping the demand of mixed-criticality sporadic tasks, *Proc. Int. Euromicro Conf. Real-Time Syst.* (2012), pp. 135–144.
23. D. Zhu, R. Melhem and D. Mosse, The effects of energy management on reliability in real-time embedded systems, *Proc. Int. Conf. Computer-Aided Design* (2004), pp. 35–40.
24. S. Aminzadeh and A. Ejlali, A comparative study of system-level energy management methods for fault-tolerant hard real-time systems, *IEEE Trans. Comput.* **60** (2011) 1228–1299.
25. L. Sha, J. P. Lehoczky and R. Rajkumar, Solutions for some practical problems in prioritizing preemptive scheduling, *Proc. Int. Symp. Real-Time Systems* (1986).
26. J. Smolens, B. Gold, J. Kim, B. Falsafi, J. Hoe and A. Nowatryk, Fingerprinting: Bounding soft-error-detection latency and bandwidth, *ACM SIGPLAN Not.* **39** (2004) 224–234.
27. P. Axer, M. Sebastian and R. Ernst, Reliability analysis for MPSoCs with mixed-critical, hard real-time constraints, *Proc. Int. Conf. Hardware/Software Codesign and System Synthesis* (2011), pp. 149–158.
28. C. Gu, N. Guan, Q. Deng and W. Yi, Partitioned mixed-criticality scheduling on multiprocessor platforms, *Proc. Int. Conf. Design, Automation and Test in Europe* (2014), pp. 1–6.