Scheduling Mixed-Criticality Real-Time Tasks with Fault Tolerance

Jian (Denny) Lin[1], Albert M. K. Cheng[2], Douglas Steel[1], Michael Yu-Chi Wu[1]

[1]Department of Management Information Systems, University of Houston - Clear Lake, Houston, TX 77058 USA
[2]Department of Computer Science, University of Houston, Houston, TX 77024 USA

***Abstract* - Enabling computer tasks with different levels of criticality running on a common hardware platform has been an increasingly important trend in the design of real-time and embedded systems. On such systems, a real-time task may exhibit different WCETs (Worst Case Execution Times) in different criticality modes. It has been well-known that traditional real-time scheduling methods are not applicable to ensure the timely requirement of the mixed-criticality tasks. In this paper, we study a problem of scheduling real-time, mixed-criticality tasks with fault tolerance. An off-line algorithm is proposed to enhance the performance of the system when it runs into a high-criticality mode from a low-criticality mode. A novel on-line slack-reclaiming algorithm is also proposed to recover from as many faults as possible before the jobs' deadline. Our simulations show that an improvement of about 30% in performance can be seen between our algorithm and a regular slack-reclaiming method.**

I. Introduction

THE integration of multiple functionalities on a single hardware platform is an increasing trend in the design of embedded systems with the consideration of reducing cost. While the tasks running on these systems share resources they do not share the same importance (criticality). Therefore, the concept of *mixed-criticality* has risen. Some widely-discussed cases about mixed-criticality are the application domains that need to be certified correct by Certification Authorities (CA's) [1]. In these cases, different computer tasks performed on the same system require different levels of assurance. This difference probably produces different WCETs of the the critical tasks estimated by the CA's and the system designers. The CA's may be more concerned about some tasks' assurance than the system designers do. As a result, it brings issues to the scheduling of real-time tasks with different levels of criticality in a timely manner. It has been well-known that conventional scheduling methods cannot satisfactorily address these issues.

Mixed-criticality systems recently become one of the research focuses in the community of real-time and embedded systems. For examples, Sanjoy Baruah *et al.* demonstrate the intractability of determining whether a mixed-criticality system can be scheduled to meet all its certification requirements, and then two scheduling techniques are proposed to scheduling such mixed-criticality systems [1]. De Niz *et al.* study the *criticality inversion* problem and propose a new scheduling scheme called *zero-slack scheduling* which can be used with priority-based preemptive schedulers (e.g., RMS) [2]. Later, the work is extended to work with solutions for a distributed system [3]. For solving the problems under a dynamic priority scheduler EDF (Earliest Deadline First), Baruah *et al.* [4] propose an effective and efficient scheduling algorithm, namely EDF-VD (virtual deadline), in which two different deadlines are used for some tasks if they may exhibit two different WCETs during the run-time. In order to guarantee the timeliness of high-criticality tasks most of existing algorithms completely sacrifice the executions of low-criticality tasks [1-6]. This strategy is too conservative and not necessary in most cases. Also, too many jobs abandoned can seriously degrade the system's performance or even cause service abrupt.

Real-time systems not only have temporal constraints to meet, computation quality constraints are also clearly important for a system with critical tasks. In a mixed-criticality system, faults or errors may happen during tasks' execution which can either produce incorrect results or cause critical tasks to miss deadlines, both of which may be catastrophic. It has been shown that in a computer system transient faults occur much more frequently than permanent faults do [7, 8]. Transient faults can be tolerated by adding redundancy where a task will be re-executed if it completes with errors. There is little work to study both fault-tolerance and mixed criticality systems. In [14], the author studies the fixed-priority schedulablity test condition for a mixed-criticality system. In [15], Huang et al describes a method to convert the fault-tolerant problem into a standard scheduling problem in a mixed-criticality system.

In this work, we consider a problem that schedules a set of real-time, fault-tolerant tasks in a mixed-criticality system using EDF. Each task in the system is periodic and characterized by a 4-tuple of parameters: $\tau_i = (p_i, X_i, c_i(LO), c_i(HI))$. Without loss of generality, all tasks are assumed to be active at time 0. The p_i is a period which is the length of the interval between any two τ_i's consecutive job releases and also the relative deadline of the task. The $X_i \in \{LO, HI\}$ denotes the criticality of τ_i. A HI-criticality task τ_i may exhibit two WCETs $c_i(LO)$ and $c_i(HI)$ during the run-time where $c_i(HI) \geq c_i(LO)$. A LO-criticality task τ_i has only the $c_i(LO)$ defined and its $c_i(HI)$ is defined as *none*. After a system starts to run, all tasks have an infinite sequence of jobs to execute. The jth job of τ_i is released at $(j-1)p_i$. Initially, all HI-criticality and LO-criticality tasks in the system are scheduled using their $c_i(LO)s$ and in this stage the system is said to be in the LO-criticality mode. During the execution, a HI-criticality task may signal that its execution time exceeds its $c_i(LO)$. At this point, all HI-criticality tasks will assume

[2]Supported by the National Science Foundation under Awards No. 0720856 and No. 1219082.

their $c_i(HI)s$ and the system will go into the HI-criticality mode. In order to enhance the system's reliability, the primary and re-execution approach [9, 10] is used and a re-execution may be performed after errors are detected at the completion of the primary execution.

II. An Off-Line Algorithm

In our work, we adopt similar notations as used in [4] for utilization parameters for tasks. A general format is defined as follows:

$$U_x^y(a) = \sum_{a \in \{pri,re\} \wedge X_i = x} \frac{c_i(y)}{p_i} \quad (1)$$

The superscript and the subscript next to the U denote the system mode and type of task, respectively. If with a, it indicates that it is for primary execution or re-execution.

In [4], without a consideration of primary or re-execution, a virtual deadline used for each HI-criticality task in the LO-criticality mode is obtained off-line with a scaling factor x, where the virtual relative deadline of the HI-criticality τ_i is equal to $x \times p_i$. The x is a value defined as:

$$x \in [\frac{U_{HI}^{LO}}{1 - U_{LO}^{LO}}, \frac{1 - U_{HI}^{HI}}{U_{LO}^{LO}}] \quad (2)$$

Any value in the range can be used for x. When a system is signaled with the HI-criticality mode, all HI-criticality tasks τ_i use their $c_i(HI)$ and all LO-criticality tasks are abandoned immediately. If such a range is not found, for example, $\frac{U_{HI}^{LO}}{1-U_{LO}^{LO}} > \frac{1-U_{HI}^{HI}}{U_{LO}^{LO}}$, the HI-criticality tasks cannot be guaranteed to meet their deadline during the mode conversion.

The aspect of our work is different. We assume that the low-criticality tasks are not trivial but the reliability of them can be traded for schedulability. In our method, we not only assure the primary execution and re-execution for all HI-criticality tasks, but also maximize the executions of LO-criticality tasks. If the resource is sufficient, the re-executions for LO-criticality tasks are scheduled as well in order to enhance the overall reliability.

In this section, we introduce an off-line algorithm to calculate (maximize) the number of tasks, including HI- and LO-criticality tasks, with the primary and re-executions reserved. The system starts with every task having both primary and re-executions reserved in the LO-criticality mode. HI-criticality tasks are required to reserve their primary execution and re-execution in HI-criticality modes to maintain high reliability. Next, we try to reserve the primary executions then the re-executions of LO-criticality tasks. During the HI-criticality mode, the primary and re-executions of the HI-criticality tasks have the equal importance and they must be reserved, but for LO-criticality tasks the primary executions are reserved before the re-executions. The re-executions of LO-criticality tasks are reserved only after all other primary executions are reserved. The order to reserve the executions is as follows: primary and re-executions of HI-criticality tasks, primary execution of LO-criticality tasks and then re-execution of LO criticality tasks. For the problem to be non-trivial, we assume that the re-executions cannot be guaranteed for all LO-criticality tasks in the HI-criticality mode.

According to the schedulability test of EDF, for the system schedulable in the LO-criticality mode with every task's primary and re-execution reserved, the total utilization is at most 1:

$$U_{HI}^{LO}(pri) + U_{HI}^{LO}(re) + U_{LO}^{LO}(pri) + U_{LO}^{LO}(re) \leq 1 \quad (3)$$

During the execution, the system goes into the HI-criticality mode if there is any HI-criticality task exceeding its $c(LO)$. We call the event as mode conversion. In a mode conversion, all HI-criticality tasks must maintain their fault-tolerance and both of their primary and re-execution will use their $c(HI)$ for the WCET. In order to walk through the mode conversion, the x is calculated as in (2), where $x \times d_i$ is used for the HI-criticality task's relative deadline in LO-criticality mode. It switches back to the original relative deadline while the system goes into the HI-criticality mode. The following conditions calculate the two boundaries in (2) for the scaling factor to ensure that all HI-criticality tasks' primary and re-execution are reserved in the HI-criticality mode without missing their deadlines:

$$x \geq \frac{U_{HI}^{LO}(pri) + U_{HI}^{LO}(re)}{1 - U_{LO}^{LO}(pri) - U_{LO}^{LO}(re)} \quad (4)$$

$$x \leq \frac{1 - (U_{HI}^{HI}(pri) + U_{HI}^{HI}(re))}{U_{LO}^{LO}(pri) + U_{LO}^{LO}(re)} \quad (5)$$

An important observations in above inequalities is that there is a room between these two boundaries so that it can be used to keep some LO-criticality tasks' executions in the system's HI-criticality mode. The significance of utilizing this room is two-fold. First, having more executions of LO-criticality tasks performed increases a system's overall value and improves the system's performance. Second, when more tasks have a re-execution reserved it makes the system more reliable and predictable. With doing a small modification, the inequalities 4 and 5 can be re-written for including the reserved re-executions for some LO-criticality tasks in the HI-criticality mode:

$$x \geq \frac{U_{HI}^{LO}(pri) + U_{HI}^{LO}(re) + U_{LO}^{LO}(pri)'}{1 - U_{LO}^{LO}(pri)'' - U_{LO}^{LO}(re)} \quad (6)$$

$$x \leq \frac{1 - (U_{HI}^{HI}(pri) + U_{HI}^{HI}(re) + U_{LO}^{LO}(pri)')}{U_{LO}^{LO}(pri)'' + U_{LO}^{LO}(re)} \quad (7)$$

The $U_{LO}^{LO}(pri)'$ in inequality 6 and the $U_{LO}^{LO}(pri)''$ in inequality 7 denote, of the LO-criticality tasks, the utilization for the reserved primary execution and the utilization of the primary execution not reserved, respectively. If the resource sufficiently allows all LO-criticality tasks to have their primary executions reserved, we can try to reserve more re-executions for LO-criticality tasks.

$$x \geq \frac{U_{HI}^{LO}(pri) + U_{HI}^{LO}(re) + U_{LO}^{LO}(pri) + U_{LO}^{LO}(re)'}{1 - U_{LO}^{LO}(re)''} \quad (8)$$

$$x \leq \frac{1 - (U_{HI}^{HI}(pri) + U_{HI}^{HI}(re) + U_{LO}^{LO}(pri) + U_{LO}^{LO}(re)')}{U_{LO}^{LO}(re)''} \quad (9)$$

The $U_{LO}^{LO}(re)'$ in inequality 8 and the $U_{LO}^{LO}(re)''$ in inequality 9 denote, of the LO-criticality tasks, the utilization for the reserved re-execution and the utilization of the re-execution not reserved, respectively.

We notice that when an execution of a LO-criticality task is moved from a denominator to a numerator, The gap between the two boundaries is narrowed. A similarity can be found between the problem and the bin-packing problem. Intuitively, the smallest first approach is a good way to optimize the solution. We demonstrate our solution in Algorithm 1 (Max. Re-executions). In the input, U_1 represents the numerator of inequality 6 or 8, U_2 represents the sub-tractor in the numerator of inequality 7 or 9, and U_3 is equal to the remaining, total utilization for un-reserved executions. The x_1 and x_2 are the left and right boundaries of x and when the final boundaries are found we set $x = x_2$. Utilizations of LO-criticality tasks that are not reserved are sorted from small to large as part of the input. The primary executions are allocated for reservations before the re-executions. Line 8 and Line 9 tell that before the loop starts if $x_2 < x_1$, the system is not able to reserve all HI-criticality tasks' primary executions and re-executions. From Line 11 to Line 24, the loop adds one primary execution of a LO-criticality execution a time, from small to large, to the reserved ones. If all LO-criticality primary executions can be reserved, it will try to reserve more LO-criticality re-executions. It calculates x_1 and x_2 in each iteration by using the updated utilizations. The x is returned while the gap between x_1 and x_2 is minimized. If a system is schedulable with the virtual deadlines, during LO-criticality mode the value of $x \times d_i$ is assigned as the relative deadlines to all HI-criticality tasks for both primary and re-executions, and to all LO-criticality tasks for their reserved executions. The executions from LO-criticality tasks that are not reserved, use their original deadlines in the LO-criticality mode and are abandoned or run in background in the HI-criticality mode.

Compared with using the approaches discarding all LO-criticality tasks, Algorithm 1 not only guarantees the HI-criticality tasks' deadlines and fault-tolerance in both HI and LO-criticality modes, some LO-criticality tasks' executions are also reserved if the resource is sufficient. The guarantee of meeting the reserved executions' deadlines during both criticality modes and the mode conversion is proved in Theorem 1.

$\boldsymbol{Theorem 1}$: There is no deadline miss if it uses the x returned by Algorithm 1 to calculate the virtual deadlines used in LO-criticality mode for all reserved executions and discard the un-reserved re-executions during the HI-criticality mode.

The proof is supported by two facts. The first one is the mapping of our task set to the task set used in the proof in [4] to support the validity using virtual deadlines. Algorithm 1 in fact makes the following executions' behavior as HI-criticality: primary and re-executions of HI-criticality tasks, and reserved LO-criticality executions. By using the calculated virtual deadlines, every task's reserved execution does not miss its deadline. The correctness directly follows the Theorem 1 and Theorem 2 in [4]. The second fact is that by using EDF, any reduction of execution does not cause a deadline violation. In our problem, if a primary execution completes correctly, its reserved re-execution is not performed. This is the same as reducing the re-execution's running time to be zero which will not cause any deadline violation.

Input : τ, Γ, ξ, U_1, U_2, x_1, x_2;
Output: the scaling factor x;
1 initialization:
2 $\Gamma = \tau_{HI}(pri) \cup \tau_{HI}(re)$, $\xi = \tau_{LO}(pri) \cup \tau_{LO}(re)$
3 $U_1 = U_{HI}^{LO}(pri) + U_{HI}^{LO}(re)$
4 $U_2 = U_{HI}^{HI}(pri) + U_{HI}^{HI}(re)$
5 $U_3 = U_{LO}^{LO}(pri) + U_{LO}^{LO}(re)$
6 ξ is sorted from small to large using utilization for the primary and re-executions, respectively. The primary executions are ordered before the re-executions.
7 $x_1 = \frac{U_1}{1-U_3}$, $x_2 = \frac{1-U_2}{U_3}$, x=x_2;
8 **if** $(x_2 < x_1)$ **then**
9 | not schedulable;
10 **else**
11 | **while** $|\xi| > 1$ **do**
12 | | U_i = the first execution's utilization in ξ ;
13 | | $U_1 = U_1 + U_i$;
14 | | $U_2 = U_2 + U_i$;
15 | | $U_3 = U_3 - U_i$
16 | | $x_1 = \frac{U_1}{1-U_3}$, $x_2 = \frac{1-U_2}{U_3}$;
17 | | **if** $x_1 \le x_2$ **then**
18 | | | $\Gamma = \Gamma \cup \tau_{LO}(U_i)$;
19 | | | x = x_2;
20 | | | $\xi = \xi - \tau_{LO}(U_i)$;
21 | | **else**
22 | | | return x;
23 | | **end**
24 | **end**
25 | return x;
26 **end**

Algorithm 1: Maximize Re-executions

	p	X	c(LO)	c(HI)	x	d_{pri}	d_{re}
τ_1	30	HI	3	4.5	0.8	24	24
τ_2	100	HI	5	12	0.8	80	80
τ_3	200	LO	10	none	0.8	160	160
τ_4	50	LO	3	none	0.8	40	50
τ_5	50	LO	7	none	0.8	40	50

Table 1. Calculated relative deadlines of primary and re-executions of a 5-tasks set

Table 1 gives an example of using Algorithm 1 to calculate the virtual deadlines for a task set of 5 tasks. In the task set, there are two HI-criticality tasks and three LO-criticality tasks. It is not possible to schedule all tasks for guaranteeing both of their primary and re-executions during the HI-criticality mode because the total utilization is larger than 1. Instead of discarding all LO-criticality tasks in the HI-criticality mode, we can ensure all of the primary executions for all tasks and the re-executions of τ_1, τ_2 and τ_3. The calculated virtual deadlines are shown in the table.

III. An On-Line Slack Reclaiming Algorithm

The strategy of using re-executions to preserve the reliability is relatively conservative. It is expected that most job instances in a system do not produce errors and therefore the resource for the reserved re-executions is wasted. Also, it is well adopted that in most of the time a real-time task does not use up its WCET. Clearly, slack is generated in these two cases. When a fault is detected by a LO-criticality job without a re-execution reserved, the slack may not be large enough to re-execution. However, if we accept a small probability of a sacrifice of the predictability, we may be able to borrow some slack from a future execution to increase the amount of the available slack at that time. Figure 1 shows an example of

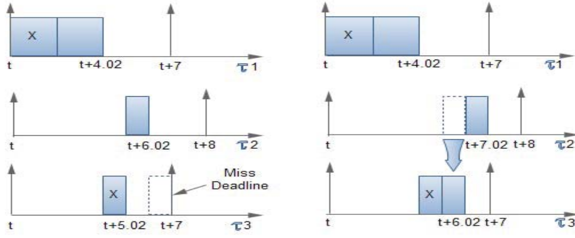

Fig. 1. Borrowing another job's future re-execution to the current re-execution

three tasks in the HI-criticality mode: τ_1 is a HI-criticality task with $c_1(HI) = 2.01$, and τ_2 and τ_3 are two LO-criticality tasks with the same $c(LO) = 1$. Both τ_1 and τ_2 have a re-execution reserved. Assume that time *t* is a LCM (Least Common Multiple) of the three tasks and their periods are 7, 8 and 7. So, for each task there is a job arriving at *t*. The HI-criticality mode is assumed to be the mode at and after time *t*. Now suppose that both the first jobs of τ_1 and τ_2 in the figure have a fault detected at their completion. The left schedule shows that the first job of τ_3 will miss its deadline at time t+7 if a re-execution is performed using the slack generated from the no-operation of τ_2's re-execution. The right schedule shows a possible solution that a future re-execution of τ_2 can be borrowed to re-execute the job of τ_3 early. If the job of τ_2 completes without errors, no impact is posed to it. Even if the job of τ_2 ends with an incorrect result due to a lack of the re-execution, the consequence is not catastrophic because both jobs are not HI-criticality. Considering the small likelihood of a fault actually occurring, the idea behind this solution has the potential to increase the overall reliability and system's performance.

Now, we are ready to present our on-line slack reclaiming algorithm. When an additional re-execution not reserved is needed to perform, the slack generated from the early completion of jobs (including no-operations of reserved re-executions) and system idle time are firstly used because they are most safe. A borrowing from the future is performed only if the slack in the previous cases is not sufficient. The borrowing only happens at a future re-execution of a LO-criticality task because HI-criticality tasks require the highest level of fault-tolerance. Algorithm 2 is based on the approach in [5, 6] (CASH) and a modification to allow the borrowing of future re-executions is applied.

1) Each task τ_i is associated with a server S_i characterized by a current budget b_i and an ordered pair (q_i, p_i), where q_i is the maximum budget and p_i is the period of the server and the task. If a task has a re-execution statically reserved, $q_i = 2 \times c_i$; otherwise, $q_i = c_i$.
2) When the system in the LO-criticality mode, the c_i for a HI-criticality task is the LO one, and the p_i is equal to its virtual relative deadline. If a mode conversion occurs, all HI-criticality tasks will use their HI-criticality execution time for c_i and their original relative deadline as p_i.
3) At each time $(k-1) \times p_i, k \in \mathbb{N}^+$, the server budget b_i is recharged at the maximum value q_i and the new deadline of the server is $d_{i,k} = k \times p_i$. The $\tau_{i,k}$ with a total execution time equal to b_i is inserted into a waiting queue, and scheduled using the server's deadline.
4) A server S_i is said to be active at time *t* if there are jobs associated with it pending; otherwise, it is said to be idle.
5) There is a slack queue such as the one used in [11]. Whenever $\tau_{i,k}$ is scheduled for execution, it always uses the slack with the earliest deadline such that $d_q \le d_{i,k}$. Otherwise, its own budget b_i is used. When $\tau_{i,k}$ is executed, the slack in the slack queue or the server budget is decreased by the same amount. When a slack capacity in the queue becomes zero, it is removed from the queue.
6) When a fault of a completion of $\tau_{i,k}$'s primary execution is detected at *t*, the total execution time of $\tau_{i,k}$ is increased by c_i.
7) When the server S_i is active and b_i becomes zero at *t*, it finds the first LO-criticality job with an execution reserved in the queue which has not finished its primary execution (if any). Then, do the following:
   a) The job's server donates a budget of its re-execution to S_i,
   b) S_i's deadline is set to $d_d - c_d^{'}$, where d_d is the deadline of the donating server of the job and $c_d^{'}$ is the remaining primary execution time of the server from the donating task.

   If no such LO-criticality jobs in the waiting queue exist, the re-execution of $\tau_{i,k}$ only runs while the system becomes idle.
8) When a job's deadline is reached, even though it has not been finished yet, the job is terminated and its new instance in the next interval starts as in step 3.
9) When a job finishes or is terminated, the residual budget of its server, if any, is inserted into the slack queue using its server's deadline.
10) Whenever the system becomes idle for an interval of time, the slack with the earliest deadline, if any, in the slack queue is decreased by the same amount of time until the queue becomes empty.

Algorithm 2: CBS-FT (CBS-Fault Tolerance)

The step 7 states how the server S_i borrows and uses a

capacity of time from a donating server by setting a new deadline, $d_d - c_d^{'}$, which is the latest time to use that capacity. We restrict to use our on-line algorithms in the cases where all LO-criticality tasks have their primary executions reserved. Otherwise, we just execute the un-reserved executions using the regular slack.

We use the same example in Figure 1 to explain the process. At time t+5.02, a fault is detected for τ_3 and another WCET (1.0) is added to the execution time requirement. At this point S_3's server budget is equal to zero and it finds S_2 from the waiting queue. S_2 donates a capacity of time of 1.0 to S_3 so that now $b_3 = 1$ and $b_2 = 1$. S_3's new deadline is set to be t+8-1=t+7 so τ_3 is scheduled before τ_2 and the re-execution of τ_3 completes in time. Later, τ_2 completes its primary execution correctly and the budget for its reserved re-execution is not wasted. In this case, all tasks finally complete with the correct results.

IV. Perfomance and Discussions

In this section, we perform simulations to evaluate the performance of our off-line Max. Re-executions and on-line CBS-FT algorithm. For the algorithm Max. Re-execution, each time a task set of ten periodic tasks is randomly generated. The total utilization of the task set is larger than 1.0 so that not all tasks have their primary and re-executions reserved. In these ten tasks, four are HI-criticality and the other six are LO-criticality. We apply the off-line Max. Re-execution algorithm to the task set so that the reserved executions for the LO-criticality tasks can be maximized. We perform this experiment 100 times and each time the number of reserved primary and re-execution for the LO-criticality tasks are recorded. The results are shown as in Figure 2 and Figure 3. It can be seen that instead of discarding the executions of all LO-criticality tasks, we can nicely reserve a great number of their primary executions. The average is 4.26 of 6 LO-criticality tasks. Since we reserve a re-execution only if all primary executions of the LO-criticality tasks are reserved, the average number of the reserved re-execution is relatively small as 1.32 task per 6 tasks. Despite, there are about 40% of the cases in which at least one LO-criticality task becomes fault-tolerant during the HI-criticality mode.

While all primary executions are reserved, we can use the *borrow from future* technique as implemented in Algorithm 2. A task set of five periodic tasks is generated randomly and the task set is selected if all tasks have their primary execution reserved. Among the five tasks, two are randomly selected as HI-criticality tasks and the rest are LO-criticality tasks. Algorithm 1 is used to determine which re-executions are reserved. A valid test case used to evaluate the Algorithm 2 should contain at least one LO-criticality task with a re-execution reserved. The tasks' periods range from 30 to 200 and they are scheduled within an interval of one million time units. Within that interval each time the total number of job instances generated is expected between 60,000 to 100,000. A fault occurrence rate between 0.05 and 0.5 is used to control whether or not a job completes with a fault in its primary execution. A fault is recorded if the job's re-execution does

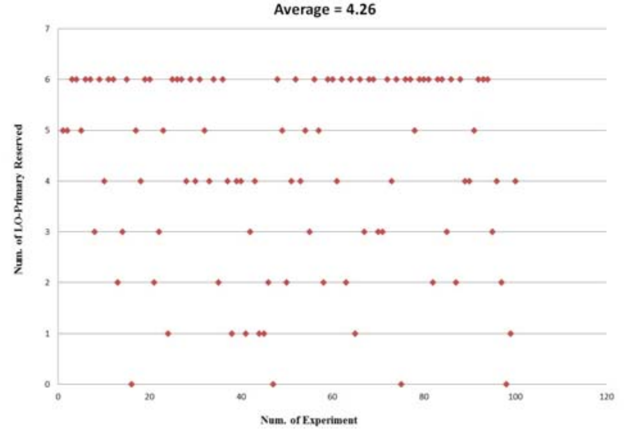

Fig. 2. The Number of Reserved Primary Executions using Algorithm 1

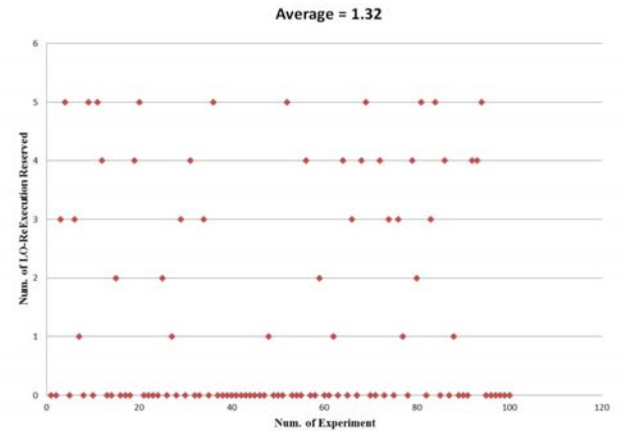

Fig. 3. The Number of Reserved Re-Executions using Algorithm 1

not successfully complete. For each experiment, we perform it 10 times using the same parameters and the average is used for our results.

In the experiments, the fault rate has a big impact to our results and another factor is the lower bound of each job's actual execution time. The smaller the fault occurrence rate/the lower bound of the actual execution times, the larger amount of regular slack that can be used to recover faults and the lower risk of missing deadline for the jobs lending their re-execution slack. We investigate the effects of these two factors in our experiments and the results are shown in Table 2 and Table 3. In Table 2, each row is a set of data for a specific fault rate. The data compares of a slack reclaiming method using the regular slack only (no borrow from future) and our CBS-FT algorithm. The number of final faults are recorded when these two approaches are used. The percentages of faults recovery are also calculated. It can be seen that the CBS-FT with the borrowing of future slack always outperforms the approach using regular slack only. If we compare the number of faults recorded, an improvement of about 30% in performance can

Fault Rate	Faults in Primary Execution	Faults Recorded (Regular Slack)	Faults Recovered (Regular Slack)	Faults Recorded (CBS-FT)	Faults Recovered (CBS-FT)	Faults on Jobs w/ Re-execution Originally Reserved
0.05	4,379	1,178	73%	843	81%	0.00%
0.2	17,647	4,989	72%	3,611	80%	0.14%
0.3	26,554	7,944	70%	5,484	79%	0.5%
0.4	35,324	10,905	69%	7,685	78%	1.42%
0.5	43,832	14,074	68%	9,868	77%	2.95%

Table 2. Experimental results based on the fault occurrence rate (lower bound of actual execution times is 1.0)

Execution Times' Range	Faults in Primary Execution	Faults Recorded (Regular Slack)	Faults Recovered (Regular Slack)	Faults Recorded (CBS-FT)	Faults Recovered (CBS-FT)	Faults on Jobs w/ Re-execution Originally Reserved
0.9 - 1.0	43,298	13,935	68%	9,715	78%	2.20%
0.8 - 1.0	44,021	13,401	70%	9,366	79%	0.4%
0.7 - 1.0	43,589	11,887	73%	8,319	81%	0.06%
0.6 - 1.0	43,420	10,816	75%	7,607	82%	0.00%
0.5 - 1.0	43,489	9,922	77%	7,022	84%	0.00%
0.2 - 1.0	44,019	5,291	88%	3,828	91%	0.00%

Table 3. Experimental results based on the lower bound of actual execution times of jobs (fault-rate = 0.5)

be seen by using our CBS-FT.

With using our algorithm, it is possible that a LO-criticality task originally having its re-execution reserved violates its deadline because it lends its budget to another task to recover from fault. The last column in the table shows the percentages of the faults caused by this scenario. As we explained it earlier, since later jobs have a bigger chance of not executing the re-execution or using the regular slack generated later, a borrowing of slack from future re-execution does not significantly degrade the overall performance a lot. While the fault rate is small, nearly no lending jobs miss their deadline for recovering from their faults. Even if the fault rate is as high as 0.5, the number of faults on the jobs with the re-execution originally reserved is relatively small so that the system's performance is well maintained. Table 3's results are similar, based on the lower bounds of the actual execution times. The results are obtained by using a fixed fault occurrence rate of 0.5. A specific range that represents the lower bound of the actual execution time is used in each set. For example, 0.5 - 1.0 means that the actual execution time is between 50% and 100% of the WCET, using an even distribution. While tasks are more likely to use less time for execution, more faults can be recovered by using both approaches but our CBS-FT always does the job better. Also, while the lower bound of the actual execution time becomes lower and lower, the difference of the fault recovery rate between the two approaches becomes smaller and smaller. This is because when the jobs complete earlier, their slack can be used by other jobs earlier. While more jobs use the regular slack to recover faults, fewer jobs need to borrow slack from the future re-executions.

V. Discussion and Summary

This paper studies a novel problem of scheduling a set of mixed-criticality, real-time tasks and considering the fault-tolerance issue. Two algorithms are proposed. The static algorithm works for maximizing the reserved executions of the LO-criticality tasks including their primary and back-up executions. The on-line algorithm improves the regular approach by exploiting a technique to borrow slack from future executions. The evaluation results show that both of our algorithms significantly improve the performance. For the sake of simplicity, we assume that the system has two levels of criticality but the results can be generally expanded to have multiple levels. Also, we assume that using one re-execution sufficiently satisfies the requirement of fault-tolerance. How to handle with multiple number of faults will become the target to extend this paper. Finally, another common technique used in fault-tolerance is checkpointing. It will be an interesting problem how to use checkpointing in mixed-criticality systems for fault-tolerance.

References

[1] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow and L. Stougie, *Scheduling Real-Time Mixed-Criticality Jobs*, IEEE Transactions on Computers, Vol. 61, No. 8, August 2012.

[2] D. de Niz, K. Lakshmanan, and R. Rajkumar, *On the Scheduling of Mixed-Criticality Real-Time Task Sets*, in RTSS, pages 291-300, 2009.

[3] K. Lakshmanan, D. de Niz, R. Rajkumar, and G. Moreno, *Resource Allocation in Distributed Mixed-Criticality Cyber-Physical Systems*, in ICDCS pages 169178, 2010.

[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 Tas Systems*, in ECRTS, pages 147-155, 2008.

[5] S. Baruah, H. Li, and L. Stougie, *Towards the design of certifiable mixed-criticality systems*, In RTAS, pages 13-22, 2010.

[6] F. Santy. L. George, P. Thierry, and J. Goossens, *Relaxing mixedcriticality scheduling strictness for task sets scheduled with fp*. In ECRTS, pages 155-165, 2012.

[7] X. Castillo, S.R. McConnel, and D.P. Siewiorek, *Derivation and Caliberation of a Transient Error Reliability Model*, IEEE Transactions on Computers, Vol. 31, No. 7, 1982.

[8] R.K. Iyer and D.J. Rossetti, *A Measurement-Based Model for Workload Dependence of CPU Errors*, IEEE Transactions on Computers, Vol. 35, No. 6, 1986.

[9] S. Ghosh, R. Melhem and D. Mosse, *Fault-tolerant scheduling on a hard real-time multiprocessor system*, in IEEE Parallel Processing Symposium, Page 775-782, 1994.

[10] R. Al-Omari, A. K. Somani, and G. Manimaran, *Efficient Overloading Techniques for Primary-backup Scheduling in Real-Time Systems*, in Journal of Parallel Distributed Computing, Vol. 64, Issue 5, 2004.

[11] M. Caccamo, G. Buttazzo, and L. Sha, *Capacity Sharing for Overrun Control*, in RTSS, pages 295-304, 2000.

[12] C. Lin, and S. Brandt, *Improving Soft Real-Time Performance Through Better Slack Reclaiming*, in RTSS, pages 410-421, 2005.

[13] C. L. Liu and J. Layland, *Scheduling algorithms for multiprogramming in a hard real-time environment*, in Journal of the ACM, Vol 20, Issue 1, Pages 4661.

[14] R. M. Pathan, *Fault-tolerant and real-time scheduling for mixed-criticality systems*, in Real-Time Systems, Vol 50, Issue 4, pages 509-547.

[15] P. Huang, H. Yang and L. Thiele, *On the Scheduling of Fault-Tolerant Mixed-Criticality Systems*, in DAC, Pages 1-6, 2014.