# Scheduling Mixed-Criticality Real-Time Tasks in a Fault-Tolerant System

*Jian (Denny) Lin, Department of Management Information Systems, University of Houston – Clear Lake, Houston, TX, USA*

*Albert M. K. Cheng, Department of Computer Science, University of Houston, Houston, TX, USA*

*Doug Steel, Department of Management Information Systems, University of Houston – Clear Lake, Houston, TX, USA*

*Michael Yu-Chi Wu, Department of Management Information Systems, University of Houston – Clear Lake, Houston, TX, USA*

*Nanfei Sun, Department of Management Information Systems, University of Houston – Clear Lake, Houston, TX, 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 is well-known that traditional real-time scheduling methods are not applicable to ensure the timely requirement of the mixed-criticality tasks. In this paper, the authors study a problem of scheduling real-time, mixed-criticality tasks with fault tolerance. An optimal, off-line algorithm is designed to guarantee the most tasks completing successfully when the system runs into the high-criticality mode. A formal proof of the optimality is given. Also, a novel on-line slack-reclaiming algorithm is proposed to recover from computing faults before the tasks' deadline during the run-time. Simulations show that an improvement of about 30% in performance is obtained by using the slack-reclaiming method.*

*Keywords:* *Cyber-Physical Systems, Fault-Tolerance, Mixed-Criticality, Real-Time Scheduling, Real-Time Systems*

## 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 different functional tasks running on these systems share resources, they do not share the same criticality. The concept of mixed-criticality, which has been identified as one of the core foundational concepts

DOI: 10.4018/IJERTCS.2015040104

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

in the emerging disciplines of Cyber Physical Systems, has risen. A mixed-critical system is an integrated suite of hardware, operating system and middleware services and application software that supports the execution of safety-critical, mission-critical, and non-critical computer tasks within a single, secure computing platform (Barhorst et al., 2009). Each safety-critical task in these systems is characterized by a level of assurance against failure, and such a level is defined in two or more distinct safety levels. For example, IEC 61508 defining four functional safety integrity levels is an international standard, published by the International Electro-technical Commission of rules applied in industry.

In practice, a lot of safety-critical embedded systems are real-time systems because safety-critical tasks usually cannot be delayed for their execution. A correct execution of such a task needs to be done before a deadline or otherwise the consequence can be catastrophic. In real-time systems, the estimation of a task's Worst Case Execution Time (WCET) plays an important role in scheduling. When a mixed-criticality system is developed, the system designers estimate and define the WCETs as a parameter of the real-time tasks. Then, the system is validated by ensuring that the critical real-time tasks, including safety-critical and functionality-critical tasks, are schedulable. Some of the systems, such as civilian and defense avionics, are subject to mandatory certification requirements by statutory organizations (Baruah et al., 2012). These systems must be certified after the design and before their implementation. In the certification process, Certification Authorities (CA's) tend to be very concerned about the safety requirement. They may be more conservative in estimating the WCETs of the safety-critical tasks, using longer WCETs when certifying the systems. The difference between the two different estimations brings to light some of the new, interesting real-time scheduling problems. It is well-known that conventional scheduling methods cannot satisfactorily address these problems.

In a mixed-criticality system, computation quality is also clearly important. Faults or errors may happen during a task's execution which can either produce incorrect results or cause critical tasks to miss deadlines. There are two types of faults classified for happening on computer systems, permanent or transient. Permanent means faults that cannot be recovered, such as hardware damage and shutdown. Transient faults, by contrast, can be recovered after the fault is gone. A common example of transient fault is the inducing in memory cells of spurious values, caused by charged particles (e.g., alpha particles) passing through them (Krishna, 2014). In computer system transient faults occur much more frequently than permanent faults do (Castillo et al., 1982; Iyer et al., 1986). Transient faults can be tolerated by adding redundancy where a task will be re-executed if it completes with errors. In a real-time system, the redundancy is considered as a time redundancy where a re-execution is done by using slack. Slack is defined as time between when tasks finish and their respective deadlines. If the length of a slack is sufficiently long, a re-execution of the task can be run by exploiting the slack. A system that faults can be recovered generally is called a fault-tolerant system.

## RELATED WORKS

Several fundamental task models have been used to characterize real-time tasks. In periodic tasks, job instances of the same task arrive regularly with the same pace and usually each job must be finished before the next one comes. Tasks with irregular arrival times are aperiodic tasks. An aperiodic task typically has a soft deadline where occasionally missing the deadline is acceptable. Aperiodic tasks that have hard deadlines and a minimum inter-arrival time are called sporadic tasks. A variety of scheduling algorithms have been proposed to solve the real-time scheduling problems. Two classical algorithms are Earliest Deadline First (EDF) and Rate Monotonic

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

(RM) (Liu and Layland, 1973). In a queue of tasks waiting for execution, EDF is a dynamic priority scheduling algorithm that selects the task closest to its deadline to run. RM is a fixed priority scheduling algorithm that the priority is based on the periods of the tasks. The shorter the period, the higher is the task's priority. There is another important characteristic called utilization for periodic tasks. A utilization of a task is the ratio of the task's WCET and period which indicates the CPU time demand for executing the task. EDF is an optimal scheduling algorithm on preemptive, uniprocessor systems. In such a periodic system, if each task's relative deadline (the length of time between each job instance's arrival and deadline) is equal to its period, the system is schedulable if and only if the total utilization of all tasks is equal to or less than 1.0. For further information on real-time systems, please refer to the following texts (Cheng, 2002; Liu, 2000; Krishna and Shin, 1997).

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 of its certification requirements. Then, two scheduling techniques are proposed to scheduling such mixed-criticality systems (Baruah et al., 2012). 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., RM) (de Niz et al., 2009). Later, the work is extended to work with solutions for a distributed system (Lakshmanan, et al., 2010). (Baruah and Vestal, 2008) is the first paper to consider using EDF in scheduling mixed-criticality tasks. A more complete analysis of using EDF is done in (Guan et al., 2011) and (Ekberg and Yi., 2012) in which two relative deadlines are assigned to each high criticality task. One deadline is defined as the real deadline of the task and the other is an artificial earlier deadline that is used to make the high criticality tasks more likely executing before low criticality ones. A similar scheme is presented in (Baruah et al., 2008) which proposes an effective and efficient scheduling algorithm, namely EDF-VD (virtual deadline). In this scheme, the artificial deadline is obtained by a scaling factor between 0 and 1.0.

In most of the works mentioned above, a basic idea to guarantee the timeliness of high-criticality tasks when actual execution times are over their expectation is to completely sacrifice the executions of low-criticality tasks. This strategy is too conservative and probably not necessary in most cases. Too many jobs abandoned can seriously degrade the system's performance or even cause service abrupt. H. Su and D. Zhu use a different scheme (Su and Zhu, 2013) that when high-criticality tasks use less time than their WCETs on the run-time, the periods of the low criticality ones can change to run them more frequently.

There is little work to study fault-tolerance and mixed-criticality systems. In (R. M. Pathan, 2014), the author studies the fixed-priority schedulability test condition for a mixed-criticality system. In (P. Huang et al., 2014), Huang et al. describe a method to convert the fault-tolerant problem into a standard scheduling problem in a mixed-criticality system. Both works assume that low-criticality tasks are not executed if the longer WCETs are seen on the run-time.

This journal paper is an extended version of a previous work (J. Lin, et al., 2014). It is assumed that a fault-tolerant system is overloaded when longer WCETs of highly critical tasks are used. Nevertheless, it is not necessary to discard all of low criticality tasks. More specifically, the space redundancy used for fault-tolerance can be used to trade schedulability. An offline algorithm is proposed to reserve as many tasks as possible based on their WCETs. While tasks complete using less time, an online heuristic is designed to schedule more tasks on the run-time using a *best-effort* manner.

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

## PRELIMINARIES, THE DEFINITION OF THE PROBLEM, AND A MOTIVATIONAL EXAMPLE

Each task defined in this work is periodic and characterized by a 4-tuple of parameters: $T_i = (P_i, X_i, C_i(LO), C_i(HI))$. Without loss of generality, all tasks are assumed to be ready at time instant 0 and they are scheduled by EDF. The $P_i$ is the length of the interval between any two $T_i$ 's consecutive job releases and also the relative deadline of the task. A job of $T_i$ arrives at $t$ and it must complete its computation by $t + P_i$. The $X_i \in$ {LO, HI} denotes the criticality of $T_i$. A HI-criticality task $T_i$ 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 $T_i$ has only the $C_i(LO)$ defined or its $C_i(HI)$ is equal to its $C_i(LO)$. The general behavior of the system is the same as in most works of the literature (Baruah et al., 2012; de Niz et al., 2009; Lakshmanan et al., 2010; Baruah et al., 2008; Baruah et al.; 2010). After a system starts to run, all tasks may have an infinite sequence of jobs to execute. Initially, all HI-criticality and LO-criticality tasks are scheduled using their $C(LO)$ and this stage is called a LO-criticality mode. During the execution, a HI-criticality task may be detected that its execution time exceeds its $C(LO)$. At this point, it signals the system that the shorter WCETs are not trustworthy so all HI-criticality tasks will switch to use their $C(HI)$ immediately. The system is thus switched into a HI-criticality mode. In addition, the primary and re-execution approach (Ghosh, et al., 1994; Al-Omari et al., 2004) is used to enhance a system's reliability, and a re-execution may be performed after errors are detected at the completion of the primary execution. Since the likelihood of having two failed executions of the same job instance is quite low, it is assumed that using one re-execution provides good reliability. If the primary execution completes correctly, the re-execution does not run.

This work adopts the similar notations as used in (Baruah et al., 2008) for utilization parameters of tasks. A general format of a total utilization of a set of task executions is defined as follows:

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

The superscript and the subscript next to the $U$ denote the system mode and type of task, respectively. The $a$, if it exists, indicates that the task set is for primary executions or re-executions.

In (Baruah et al., 2008), it considers a dual-criticality system using two deadlines without re-executions. Besides the original deadline, a shorter, virtual relative deadline is used for each HI-criticality task running in the LO-criticality mode. When the system is switched to the HI-criticality mode, the original, longer deadline is used immediately by all of the HI-criticality tasks such that more time is given to complete the longer WCETs. The virtual deadlines are obtained off-line with a scaling factor $x$ between 0 and 1.0, and for a HI-criticality $T_i$ it is equal to $x \times P_i$. The $x$ is calculated as below.

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

If the value of $x$ is found to be a real number between 0 and 1.0, it is used to shorten a HI-criticality task's deadline in a LO-criticality mode (Baruah, et al., 2008).

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

It is assumed that both primary execution and re-execution of a HI-criticality task are required to be scheduled in both Hi-criticality and LO-criticality modes because of their high requirements in reliability. This paper studies a problem of scheduling more LO-criticality tasks. The off-line schedulability problem is named as *Max Executions* and it summarizes the requirements of the problem as follows.

1. The tasks are scheduled using EDF and the virtual deadline method is used as mentioned above.
2. HI-criticality tasks must be schedulable, including their primary executions and re-executions. Otherwise, the system is not schedulable.
3. Schedule as many primary executions of LO-criticality tasks as possible when II completes.
4. Schedule as many re-executions of LO-criticality tasks as possible when II and III complete.

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

$$U_{HI}^{LO}(pri) + U_{HI}^{LO}(re) + U_{LO}^{LO}(pri) + U_{LO}^{LO}(re) \le 1 \tag{3}$$

By defining

$$U_{HI}^{LO} = U_{HI}^{LO}(pri) + U_{HI}^{LO}(\mathrm{re})$$
$$U_{HI}^{HI} = U_{HI}^{HI}(pri) + U_{HI}^{HI}(\mathrm{re})$$
$$U_{LO}^{LO} = U_{LO}^{LO}(pri) + U_{LO}^{LO}(\mathrm{re})$$

It has

$$x \ge x_1 = \frac{U_{HI}^{LO}(pri) + U_{HI}^{LO}(re)}{1 - (U_{LO}^{LO}(pri) + U_{LO}^{LO}(re))} \tag{4}$$

$$x \le x_2 = \frac{1 - (U_{HI}^{HI}(pri) + U_{HI}^{HI}(re))}{U_{LO}^{LO}(pri) + U_{LO}^{LO}(re)} \tag{5}$$

The inequalities (4) and (5) are a straightforward extension of (2) to calculate the scaling factor for using the virtual deadlines to ensure that all HI-criticality tasks' primary and re-executions that are schedulable in the HI-criticality mode.

As an example, Table 1 lists the properties of 5 tasks, two HI-criticality ones and three LO-criticality ones, and the utilizations of each mode are calculated.

An important observation in the inequalities (4) and (5) is that there is a room between these two boundaries that can be used to schedule more LO-criticality tasks. Consider the same tasks set in Table 1. The two executions of a task $T_i$ are denoted as $T_i(pri)$ and $T_i(re)$. A LO-criticality execution can be guaranteed to complete its execution in the HI-criticality mode by practically

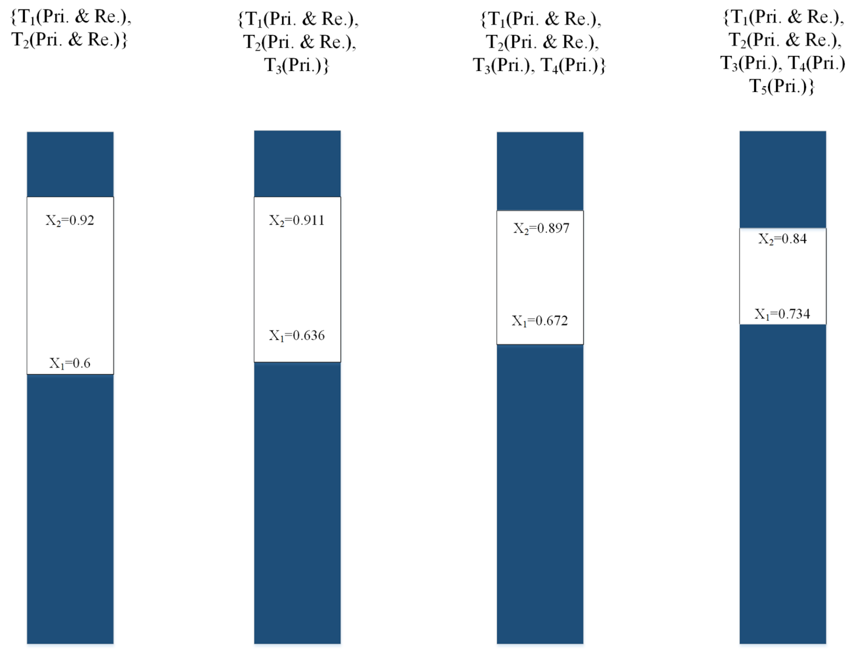
Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

*Table 1. A 5-tasks set of a mixed-criticality system*

| | **P** | **X** | **C(LO)** | **C(HI)** | $U_{HI}^{LO}(pri)\,\mathrm{or}(\mathrm{re})$ | $U_{HI}^{HI}(pri)\,\mathrm{or}(re)$ | $U_{LO}^{LO}(pri)\,\mathrm{or}(re)$ |
|---|---|---|---|---|---|---|---|
| **$T_1$** | 30 | HI | 3 | 4.5 | 0.15 | 0.27 | |
| **$T_2$** | 100 | HI | 5 | 12 | | | |
| **$T_3$** | 200 | LO | 10 | | | | |
| **$T_4$** | 50 | LO | 3 | | | | 0.25 |
| **$T_5$** | 50 | LO | 7 | | | | |

making it as a HI-criticality execution. Figure 1 shows the $x_1$s and $x_2$s calculated when LO-criticality executions are selected into a reserved set for HI-criticality tasks one by one. This process is similar to a packing process. When a task is reserved to run in the HI-criticality mode it occupies some space of a bin. The space between the two boundaries, $x_1$ and $x_2$, defines the available room that can be used to run more tasks in the HI-criticality mode. As long as the space is not empty, all tasks selected into the set are schedulable in the HI-criticality mode. In Figure 1, the leftmost bin indicates that at the beginning only the executions of HI-criticality tasks $T_1$ and $T_2$ are in the schedulable set. Then, the primary executions of $T_3$, $T_4$ and $T_5$ are added one by one into the set. It can be seen that the room in the bin is squeezed when each task is added to the set. This process illustrates that more executions of LO-criticality tasks can be reserved to complete in the HI-criticality mode.

*Figure 1. Values of the scaling factor for different tasks running in the HI-criticality mode*



Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

## AN OFF-LINE ALGORITHM

This section presents an off-line algorithm to solve the *Max Executions* problem. The idea in the solution is to execute the equivalent packing process. Each time, the LO-execution with the smallest utilization from the remaining LO-executions is selected to fill into the bin or switched to a HI-criticality execution. Then, the room of empty space in the bin defined by $x_1$ and $x_2$ is calculated. This process is repeated until no more LO-criticality executions can be reserved or converted into a HI one. The Algorithm 1 shows the details of the solution.

In the algorithm, the lower case $u_{LO}$ is used to define a utilization of an individual LO execution. Line 6 sorts the primary executions and re-executions of the LO-criticality tasks increasingly based on their utilizations, respectively. In fact, when a LO-criticality execution is added into the reserved tasks set running in the HI-criticality mode, its execution behavior is the

*Algorithm 1. Maximize Re-executions*

```
     Input  : $T$, $\Gamma$, $\xi$, $x_1$, $x_2$;
     Output: the scaling factor $x$;
 1   initialization:
 2   $\Gamma = T_{HI}(pri) \cup T_{HI}(re)$, $\xi = T_{LO}(pri) \cup \tau_{LO}(re)$
 3   $U_{HI}^{LO} = U_{HI}^{LO}(pri) + U_{HI}^{LO}(re)$
 4   $U_{HI}^{HI} = U_{HI}^{HI}(pri) + U_{HI}^{HI}(re)$
 5   $U_{LO}^{LO} = U_{LO}^{LO}(pri) + U_{LO}^{LO}(re)$
 6   $\xi$ 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   $x_1 = \frac{U_{HI}^{LO}}{1-U_{LO}^{LO}}$, $x_2 = \frac{1-U_{HI}^{HI}}{U_{LO}^{LO}}$, x=$x_2$;
 8   if ($x_2 < x_1$) then
 9       not schedulable;
10   else
11       while $|\xi| > 1$ do
12           $u_{LO}$ = the first execution's utilization in $\xi$ ;
13           $U_{HI}^{LO} = U_{HI}^{LO} + u_{LO}$;
14           $U_{HI}^{HI} = U_{HI}^{HI} + u_{LO}$;
15           $U_{LO}^{LO} = U_{LO}^{LO} - u_{LO}$
16           $x_1 = \frac{U_{HI}^{LO}}{1-U_{LO}^{LO}}$, $x_2 = \frac{1-U_{HI}^{HI}}{U_{LO}^{LO}}$ ;
17           if $x_1 \le x_2$ then
18               $\Gamma = \Gamma \cup$ the first execution in $\xi$;
19               x = $x_2$;
20               $\xi = \xi -$ the first execution in $\xi$;
21           else
22               return $x$;
23           end
24       end
25       return $x$;
26   end
          Algorithm 1: Maximize Re-executions
```

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

same as a HI-criticality task except that it has the same WCET used in both LO-criticality and HI-criticality modes. Lines 13 to 15 demonstrate how to update the total utilizations of $U_{HI}^{LO}$, $U_{HI}^{HI}$ and $U_{HI}^{LO}$ after such a conversion. Each time when a LO-execution is converted, the calculations of the two boundaries can be written as

$$x_1 = \frac{U_{HI}^{LO} + u_{LO}}{1 - U_{LO}^{LO} + u_{LO}} \tag{6}$$

$$x_2 = \frac{1 - U_{HI}^{HI} - u_{LO}}{U_{LO}^{LO} - u_{LO}} \tag{7}$$

This process is repeated until $x_1 > x_2$ when no more room available in the bin. When the process stops, the $x$ is returned. If a valid $x$ is found, the system is schedulable with using the virtual deadlines. In the LO-criticality mode the value of $x \times P_i$ is assigned as the relative deadlines to all HI-criticality tasks for their both primary and re-executions, including the ones converted from LO-criticality tasks executions. The executions from the LO-criticality tasks that are not converted use their original relative deadlines in the LO-criticality mode, and are abandoned or simply run in the background in the HI-criticality mode.

Algorithm 1 is an optimal solution of the Problem *Max Executions.* In most real-time tasks assignment problems that are modeled as a packing problem, utilizations of the tasks can be simply mapped as the items' sizes. In this paper the size occupied in the bin by a new execution is calculated by using (6) and (7). This adds an extra barrier to prove the optimality of the solution. The following theorem states the optimality of the algorithm and its formal proof is presented in the Appendix after the paper.

**Theorem 1**: Algorithm *Max Executions* is an optimized solution that maximizes the number of executions of tasks as defined in the *Max Executions Problem*.

There is no deadline miss if it uses the $x$ returned by the Algorithm 1 to calculate the virtual deadlines used in the LO-criticality mode for all reserved executions. This is supported by two facts. The first fact comes from a mapping of the reserved task set to the HI-criticality task set in (Baruah et al., 2008) with the validity of using virtual deadlines. The correctness directly follows the Theorem 1 and Theorem 2 in the paper. The second fact is supported by the sustainability property of using the preemptive EDF (Liu and Layland, 1973). Any reduction of execution time of a task does not cause a deadline violation. 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 does not cause any deadline violation.

Table 2. gives an example of using the Algorithm *Max Executions* to calculate the virtual deadlines for the task set of 5 tasks as presented in Table 1. It is not possible to schedule all of the executions during the HI-criticality mode because the total utilization is larger than 1. The Algorithm *Max Executions* is used to select additional LO-criticality executions to run in the HI-criticality mode. The value of the scaling factor returned by using the Algorithm is 0.8. The $D_{pri}$ and $D_{re}$ are the relative deadlines used in the LO-criticality mode for the reserved executions. All of the primary executions as well as the $T_3$'s re-execution is selected so their $D_{pri}$s are short-

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

*Table 2. Calculated relative deadlines of primary and re-executions of a 5-tasks set*

| | P | X | C(LO) | C(HI) | *x* | $D_{pri}$ | $D_{re}$ |
|---|---|---|---|---|---|---|---|
| $T_1$ | 30 | HI | 3 | 4.5 | 0.8 | 24 | 24 |
| $T_2$ | 100 | HI | 5 | 12 | 0.8 | 80 | 80 |
| $T_3$ | 200 | LO | 10 | None | 0.8 | 160 | 160 |
| $T_4$ | 50 | LO | 3 | None | 0.8 | 40 | 50 |
| $T_5$ | 50 | LO | 7 | None | 0.8 | 40 | 50 |

ened. The re-executions of $T_4$ and $T_5$ are not converted, and hence the system uses their original deadlines in the LO-criticality mode and may abandon them during the HI-criticality mode.

## AN ON-LINE SLACK RECLAIMING ALGORITHM

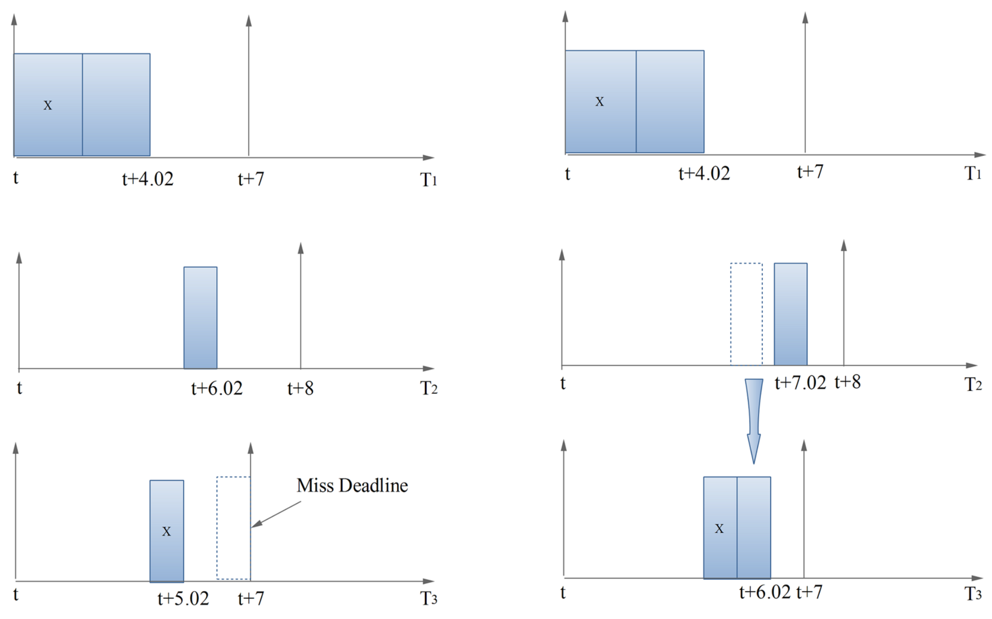
The strategy of using re-executions to preserve reliability is relatively conservative. It is expected that most job instances in a system do not produce errors and therefore most of the resource for the reserved re-executions is not used. Also, it is commonly adopted that in most of the time a real-time task uses less time than its WCET. Thus, slack can be generated from early completions of tasks and it can be used to execute un-reserved re-executions if errors happen. In some cases, it is likely that the generated slack is not sufficient to perform the re-executions. A technique of borrowing slack from future executions based on Constant Bandwidth Server (CBS) can be used to increase the present, available amount of slack. This technique has a benefit to save a task without re-execution reserved from a failure.

Figure 2. shows an example of three tasks in the HI-criticality mode: $T_1$ is a HI-criticality task with $C_1$(HI)=2.01, $T_2$ and $T_3$ are two LO-criticality tasks with the same C(LO)=1. Both $T_1$ and $T_2$ have a re-execution reserved. Assume that time *t* is a LCM (Least Common Multiple) of the three tasks' periods 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 $T_1$ and $\tau_3$ in the figure have a fault being detected at their completion. The left schedule shows that the un-reserved re-execution of $T_3$ will miss its deadline at time t+7 if it uses the slack generated from the no-operation of $T_2$'s re-execution. The right schedule shows a possible solution that a future re-execution of $T_2$ can be borrowed to re-execute the job of $T_3$ early. Because $T_2$ is not HI-criticality, the consequence of its failure is not catastrophic even if it is due to a lack of the re-execution. Considering that the likelihood of a fault to occur is relatively small, the idea behind this solution has the potential to increase the overall reliability and the system's performance.

The online solution extends and modifies the approach in (Caccamo et al., 2000; Lin and Brandt, 2005) (CASH) to allow using the borrowing of future re-executions. The key idea in the original CASH algorithm is that a task's scheduling is controlled by its server and there is a slack queue to manage slack blocks generated by the early completed or no operations of tasks. Each slack has a deadline associated with the task that generates it and a slack can be safely used before its deadline. When a task is running, its server is assigned with a time budget defined as $b_i$ and $q_i$. A task always uses the slacks, if any, having the earliest deadline before the task's deadline. If a slack is generated from a task's early completion, it will be inserted into the slack queue using the EDF order. A borrowing of slack occurs only if the general slack is not sufficient. The

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

*Figure 2. Borrowing another job's future re-execution to the current re-execution*



borrowing only happens at a future re-execution of a LO-criticality task because HI-criticality tasks require the highest level of fault-tolerance.

The Algorithm 2, named as CBS-FT, describes the extension and modification. For the details of the original CASH technique, please refer to the original paper. In step (a), if a task has its re-execution reserved, its time budget is doubled. If the task completes early or without errors, the slack including the time originally reserved for its re-execution is inserted into the slack queue with the task's deadline. In step (e), if a task that does not have a re-execution reserved fails in its primary execution, the execution time is increased by $C$ but its time budget does not change. The step (f) 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. Here, $c_d^{'}$ is the remaining execution time and $d_d$ is the deadline of the donating task's primary execution. By using the future slack with the deadline $d_d - c_d^{'}$, it is guaranteed that the execution using this slack will not cause other tasks to miss deadline. If a deadline is missed due to the borrowing, it is either in the donating task's re-execution or the task borrowing the slack. The algorithm does not use any future time of a HI-criticality task or a primary execution of any LO-criticality task.

We use the same example in Figure 2 to explain the process. At time t+5.02, a fault is detected for $T_3$ 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 $S_2$ so that now $b_3 = 1$ and $b_2 = 1$. At this point, $b_3$'s deadline is set to be *t+8-1=t+7* which happens to be its original deadline. So $T_3$'s re-execution is scheduled before $T_2$'s primary execution and the re-execution of $T_3$ completes in time. Later, $T_2$ completes its primary

*Algorithm 2. CBS-FT (CBS-Fault Tolerance)*

(a) Each task $T_i$ 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, $q_i = C_i$.
(b) 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 reserved executions will use their longer execution time (if any) for $C_i$ and their original relative deadline as $P_i$. Their $b_i$ and $q_i$ are adjusted correspondingly.
(c) 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 $T_i$ with a total execution time equal to $b_i$ is inserted into a waiting queue, and scheduled using the server's deadline.
(d) There is a slack queue such as the one used in (Caccamo et al., 2000). Whenever $T_i$ 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 a slack capacity in the queue becomes zero, it is removed from the queue.
(e) When a fault of a completion of $T_{i,k}$'s primary execution is detected at $t$, if no re-execution reserved, the total execution time of $T_{i,k}$ is increased by $C_i$.
(f) 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:
- The job's server donates a budget of its re-execution to $S_i$.
- $S_i$'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.

(g) If no such LO-criticality jobs in the waiting queue exist, the re-execution of $T_{i,k}$ only runs while the system becomes idle.
(h) When a job's deadline is reached, even though it has not been finished yet, the job is terminated.
(i) 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.
(j) 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.

execution correctly and the budget for its reserved re-execution is not wasted. In this case, all tasks finally complete with the correct results.

## PERFOMANCE AND DISCUSSIONS

Extensive simulations are performed to evaluate the performance of the off-line *Max Executions* and the on-line CBS-FT algorithms. In evaluating the off-line algorithm, it primarily focuses on how many more tasks can be safely reserved for running in the HI-criticality. For the online algorithm, a primary interest is to see the number of faults that can be recovered on-line by using the newly slack borrowing technique. The experimental results show that it is effective to use the both algorithms to successfully schedule more tasks in a mixed-criticality system.

The setting of evaluating the off-line algorithm Max Execution is explained as follows. A task set of ten periodic tasks with re-executions is randomly generated. The total utilization of the task set is larger than 1.0 and less than 1.2. Thus, it is not possible that all tasks have their primary and re-executions reserved. In these ten tasks, four are HI-criticality and six are LO-criticality tasks. The off-line *Max Execution* algorithm is applied for the task set to maximize the reserved executions of the LO-criticality tasks. This experiment is performed 100 times and each time the numbers of reserved primary and re-execution of the LO-criticality tasks are recorded. Then, the results are analyzed to see how effective the algorithm reserves LO-criticality tasks in

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

the HI-criticality mode. Figure 3 and Figure 4 show the results about reserving primary and re-executions of the LO-criticality tasks, respectively. The results of reserving primary executions are divided into seven groups as the numbers shown on the *x* axis. In these groups, 0 means that there is no any primary execution of the LO-criticality tasks to be reserved. From 1 to 6, they mean the number of primary executions of LO-criticality tasks that can be reserved. The numbers on the *y* axis indicate the number of task sets that falls into each group, or frequency, when reserving the additional primary executions. It can be seen that by using the algorithm *Max Execution* most tasks sets have more LO-criticality tasks to be reserved. In 40% of the cases, it reserves all LO-criticality tasks' primary executions. Also, some re-executions are reserved, resulting in reliability improvement. Figure 4 shows the number of re-executions of the LO-criticality tasks that are reserved in the experiments.

The Algorithm 2, CBS-FT can be used to reduce the faults during the run-time. To see the performance of the algorithm, a task set of five periodic tasks is generated randomly and 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. The Algorithm 1, *Max Executions*, is used to determine which re-executions are reserved. A valid test case used to evaluate the Algorithm CBS-FT contains at least one LO-criticality task, but not all, 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 the total number of job instances generated each time is expected between 60,000 to 100,000. A fault occurrence rate between 0.05 and 0.5 is used to control the likelihood that a fault occurs in a job's primary execution. A fault is recorded if

*Figure 3. The number of primary executions of LO-tasks reserved by using Algorithm Max. Re-execution*

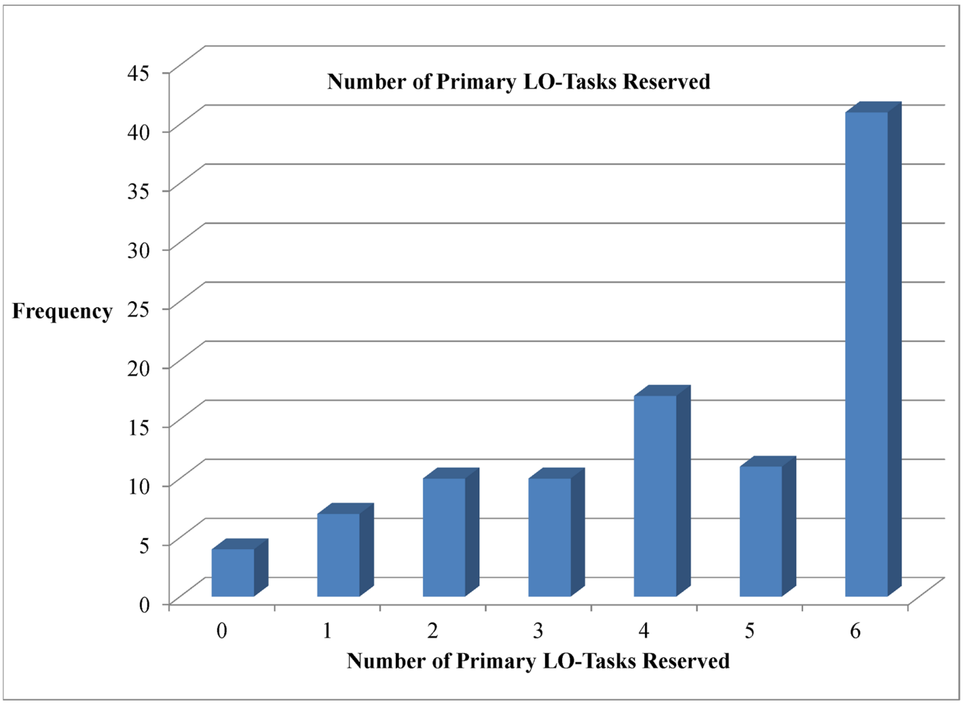
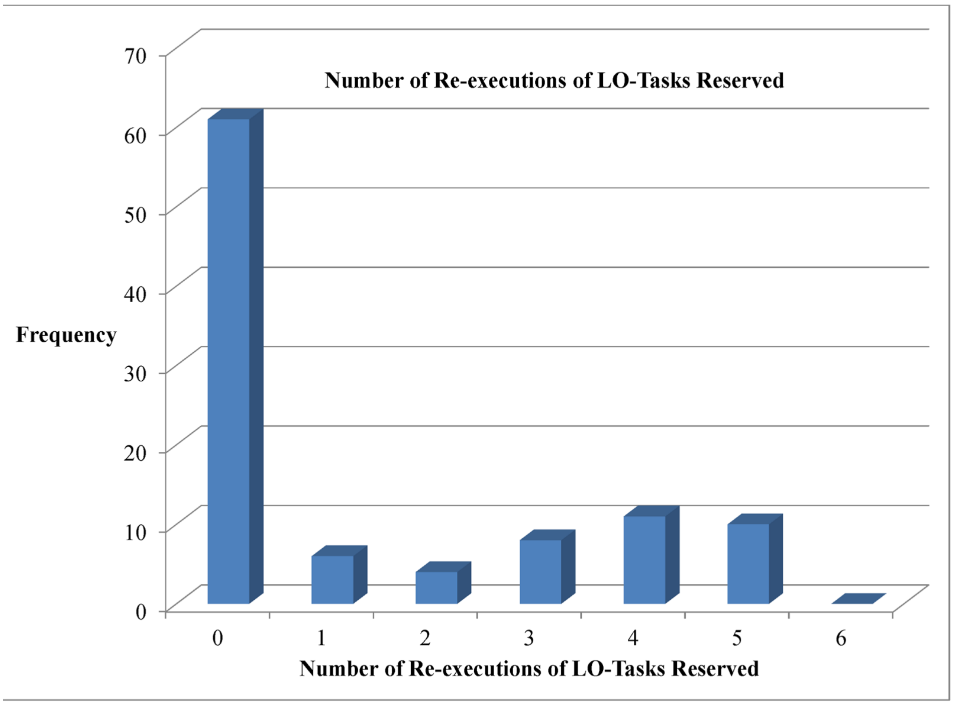


*Figure 4. The number of re-executions of LO-tasks reserved by using Algorithm Max. Re-execution*



a job's re-execution does not successfully complete. For each experiment, it performs 20 times using the same parameters and the average is used in the results.

In practice, the fault rate has a major impact to the results and another important factor is the lower bound of each job's actual execution time. The smaller the fault occurrence rate and the smaller of the actual execution times, the larger amount of regular slack that can be used to recover faults and the lower risk of missing the deadlines from the jobs that lend their re-execution slack. The effects of these two factors are investigated in the experiments and the results are summarized in Table 3 and Table 4. In Table 3, each row is a set of data for a specific fault rate. It compares a slack reclaiming method using the regular slack only (no borrowing from future) and the CBS-FT algorithm. The number of final faults are recorded when these two approaches are used. The percentages of faults recovery are also calculated. Figure 5 gives a graph of the comparisons between a regular slack-reclaim method and the CBS-FT algorithm based on the fault rates. It is assumed that each task runs in its WCET. The results show that the percentages of faults recovered by the both methods become smaller when the faults occurring rate increases. The algorithm CBS-FT steadily outperforms the regular method and an improvement of about 30% of performance can be seen. Figure 6's result is based on the actual execution time. The values on the *x* axis indicate the lower bound of the ratio of the actual execution time to the WCET. For example, 0.5 - 1.0 means that the actual execution time is between 50% and 100% of the WCET, using an even distribution. The smaller the lower end, the smaller the actual execution time

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

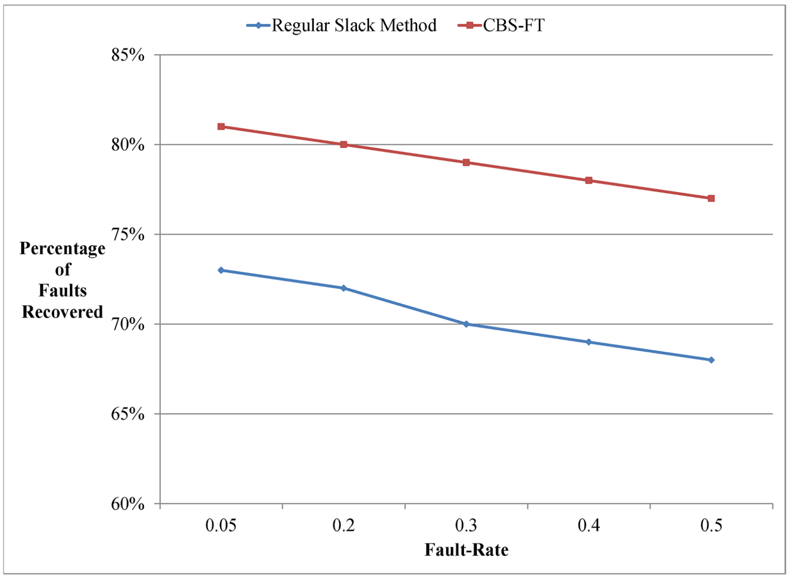
*Table 3. Experimental results based on the fault occurrence rate*

| Fault Rate | Faults in Primary Execution | Faults Recorded (Regular Slack) | Faults Recovered (Regular Slack) | Faults Recorded (CBS-FT) | Faults Recovered (CBS-FT) | Faults on lending jobs |
|---|---|---|---|---|---|---|
| 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 | 14,974 | 68% | 9,868 | 77% | 2.95% |

*Table 4. Experimental results based on the lower bound of actual execution times (fault rate = 0.5)*

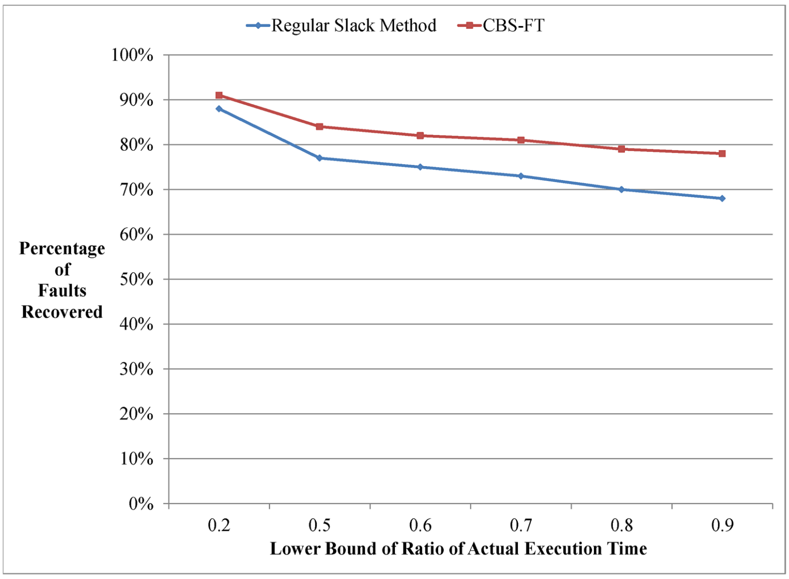
| Execution Times' Range | Faults in Primary Execution | Faults Recorded (Regular Slack) | Faults Recovered (Regular Slack) | Faults Recorded (CBS-FT) | Faults Recovered (CBS-FT) | Faults on lending jobs |
|---|---|---|---|---|---|---|
| 0.9-1.0 | 43,298 | 13,935 | 68% | 9,715 | 78% | 2.20% |
| 0.8-1.0 | 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% |

*Figure 5. Comparisons based on fault-rate*



Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

*Figure 6. Comparisons based on actual execution time*



in average. The results are obtained by using a fixed fault occurrence rate of 0.5. It can be seen that when the actual execution time is small, a lot of faults can be recovered by using both methods. Apparently, the CBS-FT algorithm keeps having a better performance than the regular slack-reclaim method. Also, while the lower bound of the actual execution time is small, the difference of the fault recovery percentages between the two compared methods is small too. This is because if the actual execution time is small, a larger amount of regular slack will be available. While more jobs use the regular slack to recover faults, fewer jobs need to borrow slack from the future re-executions.

With using the CBS-FT algorithm that borrows future slack, 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 its fault. The last column in the Table 3 and Table 4 show the percentages of the faults caused by this reason. As explained earlier, since future jobs have a big chance of not executing the re-execution or it may be able to use the regular slack generated later, a borrowing of slack from future re-executions does not significantly degrade the overall performance. While the fault rate is small, nearly no lending jobs miss their deadlines for recovering from their faults. Even if the fault rate is as high as 0.5, the number of failures on the lending jobs with the re-execution originally reserved is relatively small. The system's performance is still well maintained.

## DISCUSSION AND ACKNOWLEDGMENT

This paper studies a novel problem of scheduling a set of mixed-criticality, real-time tasks and considering the fault-tolerance issue. This section discusses some of the future works. For the sake of simplicity, the authors assume that the system has two levels of criticality but the results can be generally expanded to have multiple levels. In the use of the techniques for recovering

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

from faults, it is assumed that using one re-execution sufficiently satisfies the requirement of fault-tolerance. How to handle with multiple faults in a job execution will be the next topic. In fault tolerance, there is another popular technique: checkpointing. It becomes an interesting problem of how to use checkpointing in mixed-criticality systems for fault-tolerance. Furthermore, the algorithms can be implemented on a physical system to investigate their performance instead of using simulations.

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

## REFERENCES

Al-Omari, R., Somani, A. K., & Manimaran, G. (2004). Efficient Overloading Techniques for Primary-backup Scheduling in Real-Time Systems. *Journal of Parallel and Distributed Computing*, *64*(5), 629–648. doi:10.1016/j.jpdc.2004.03.015

Barhorst, J., Belote, T., Binns, P., Hoffman, J., Paunicka, J., Sarathy, P., . . . Urzi, R. (2009). White Paper: A Research Agenda for Mixed-Criticality Systems. Retrieved from http://www.cse.wustl.edu/~cdgill/CPSWEEK09_MCAR

Baruah, S., Bonifaci, V., D'Angelo, G., Li, H., Marchetti-Spaccamela, A., Megow, N., & Stougie, L. (2012). Scheduling Real-Time Mixed-Criticality Jobs. *IEEE Transactions on Computers*, *61*(8), 1140–1152. doi:10.1109/TC.2011.142

Baruah, S., Bonifaci, V., D'Angelo, G., Li, H., Marchetti-Spaccamela, A., van der Ster, S., & Stougie, L. (2008). The Preemptive Uniprocessor Scheduling of Mixed-Criticality Implicit-Deadline Sporadic Task Systems.*Proceedings of the IEEE Euromicro Conference on Real-Time Systems*.

Baruah, S., Li, H., & Stougie, L. (2010). Towards the design of certifiable mixed-criticality systems.*Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*. doi:10.1109/RTAS.2010.10

Baruah, S. K., & Vestal, S. Schedulability (2008). Analysis of Sporadic Tasks with Multiple Criticality Specifications. *Proceedings of the IEEE Euromicro Conference on Real-Time Systems*. doi:10.1109/ECRTS.2008.26

Caccamo, M., Buttazzo, G., & Sha, L. (2000). Capacity Sharing for Overrun Control.*Proceedings of the IEEE Real-Time Systems Symposium*. doi:10.1109/REAL.2000.896018

Castillo, X., McConnel, S. R., & Siewiorek, D. P. (1982). Derivation and Caliberation of a Transient Error Reliability Model. *IEEE Transactions on Computers*, *31*(7).

Cheng, A. (2002). *Real-Time Systems: Scheduling, Analysis and Verification. Wiley Interscience. J. Liu. (2000). Real-Time Systems*. Wiley. doi:10.1002/0471224626

Ekberg, P., & Yi, W. (2012). Bounding and Shaping The Demand of Generalized Mixed-Criticality Sporadic Task Systems.*Proceedings of the IEEE Euromicro Conference on Real-Time Systems*.

Ghosh, S., Melhem, R., & Mosse, D. (1994). Fault-tolerant scheduling on a hard real-time multiprocessor system.*Proceedings of the IEEE Parallel Processing Symposium*. doi:10.1109/IPPS.1994.288216

Guan, N., Ekberg, P., Stigge, M., & Yi, W. (2011). Effective And Efficient Scheduling of Certifiable Mixed-Criticality Sporadic Task Systems.*Proceedings of the IEEE Real-Time Systems Symposium*. doi:10.1109/RTSS.2011.10

Huang, P., Yang, H., & Thiele, L. (2014). On the Scheduling of Fault-Tolerant Mixed-Criticality Systems. *Proceedings of the ACM Design Automation Conference*. doi:10.1109/DAC.2014.6881458

Iyer, R. K., & Rossetti, D. J. (1986). A Measurement-Based Model for Workload Dependence of CPU Errors. *IEEE Transactions on Computers*, *35*(6).

Krishna, C., & Shin, K. (1997). Real-Time Systems. *McGraw-Hill*. D. de Niz, K. Lakshmanan, and R. Rajkumar. (2009). On the Scheduling of Mixed-Criticality Real-Time Task Sets. *Proceedings of theIEEE Real-Time Systems Symposium*.

Krishna, C. M. (2014). Fault-Tolerant Scheduling in Homogeneous Real-Time Systems. *ACM Computing Surveys*, *46*(4), 1–34. doi:10.1145/2534028

Lakshmanan, K., de Niz, D., Rajkumar, R., & Moreno, G. (2010). Resource Allocation in Distributed Mixed-Criticality Cyber-Physical Systems.*Proceedings of the IEEE International Conference on Distributed Computing Systems*. doi:10.1109/ICDCS.2010.91

Lin, C., & Brandt, S. (2005). Improving Soft Real-Time Performance through Better Slack Reclaiming. *Proceedings of the IEEE Real-Time Systems Symposium*.

Lin, J., Cheng, A., Steel, D., & Wu, M. (2014). Scheduling Mixed-Criticality Real-Time Tasks with Fault Tolerance. *Proceedings of the 2nd International Workshop on Mixed Criticality Systems at the Real Time Systems Symposium (RTSS)*.

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Liu, C., & Layland, J. (1973). Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, *20*(1), 46–61. doi:10.1145/321738.321743

Pathan, R. M. (2014). Fault-tolerant and real-time scheduling for mixed-criticality systems. *Real-Time Systems*, *50*(4), 509–547. doi:10.1007/s11241-014-9202-z

Su, H., & Zhu, D. (2013). An Elastic Mixed-Criticality Task Model and Its Scheduling Algorithm. *Proceedings of the ACM and IEEE Conference on Design, Automation and Test in Europe*. doi:10.7873/DATE.2013.043

Tindell, K., Burns, A., & Wellings, A. J. (1992). Mode changes in priority preemptive scheduled systems. *Proceedings of the IEEE Real Time Systems Symposium*. doi:10.1109/REAL.1992.242672

*Jian (Denny) Lin is an Assistant Professor of the Department of Management Information Systems, University of Houston – Clear Lake, USA. He received his MS and PhD, both in Computer Science, in 2006 and 2009, respectively. His research interests include but not limited to: real-time and embedded systems, fault-tolerant computing, computer networks, signal processing and simulations. Dr. Lin is a committee member of several IEEE conferences.*

*Albert M. K. Cheng is Professor and former interim Associate Chair of the Computer Science Department at the University of Houston (UH). He is the founding Director of the UH Real-Time Systems Laboratory. He received the BA with Highest Honors in Computer Science, graduating Phi Beta Kappa at age 19, the MS in Computer Science with a minor in Electrical Engineering at age 21, and the PhD in Computer Science at age 25, all from The University of Texas at Austin, where he held a GTE Foundation Doctoral Fellowship. He has served as a technical consultant for many organizations, including IBM and Shell, and was also a Visiting Professor at Rice University and the City University of Hong Kong. He is a cofounder of ZapThru.com, where he is currently the Chief Strategy and Technology Director. A recipient of numerous awards, Dr. Cheng is the author/co-author of over 220 refereed publications in leading journals and top-tier conferences, and has presented over 100 seminars, tutorials, panel positions, and keynotes. He is and has been on the technical program committees (including many program chair positions) of over 250 conferences, symposia, workshops, and editorial boards. Dr. Cheng is the author of the popular textbook entitled Real-Time Systems: Scheduling, Analysis, and Verification (Wiley). He is a Senior Member of the IEEE; an Honorary Member of the Institute for Systems and Technologies of Information, Control and Communication; and a Fellow of the Institute of Physics. He is the recipient of the 2015 University of Houston's Lifetime Faculty Award for Mentoring Undergraduate Research.*

*Douglas J. Steel is an Assistant Professor of Management Information Systems in the College of Business at the University of Houston-Clear Lake. He and his colleagues focus on the use of computer and information technology to solve business problems. Research interests include technology adoption, virtual teams and computer self-efficacy.*

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

*Michael Y.-C. Wu received the B.Sci. degree in electrical engineering in 1998, the M.Sci. degree in computer and information systems in 2000, the M.Sci. degree in electrical engineering in 2001, and the Ph.D. degree in electrical engineering in 2008, all from the New Jersey Institute of Technology in Newark, NJ (07102). Between 2008 and 2014, he worked as a software developer at several companies, including the Hewlett Packard Enterprise. Since 2014, he has been an assistant professor of Management Information Systems at the University of Houston—Clear Lake in Houston, Texas (77058), teaching various courses in programming. His current research interests include software qualities and simulations.*

*Nanfei Sun received the BEng degree in Architecture in 1994 from Chongqing University. He received the MSci degree in computer science in 2001 from University of North Carolina at Charlotte, and the PhD degree in computer science in 2006 from University of Houston. In 2005 and 2006, he worked as intern in IBM T.J. Waterson research center. Between 2007 and 2014, he worked as a software engineer in Hubwoo Inc. USA. Since 2014, he has been an assistant professor of Management Information Systems at the University of Houston—Clear Lake in Houston, Texas (77058), teaching various courses in programming and enterprise resource planning. His current research interests include ontology, data analytic, information retrieval, and computer vision.*

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

## APPENDIX

### Proof of Theorem 1

The following lemma plays a key role to prove the Theorem 1.

**Lemma 1:** Given a set of schedulable HI-criticality tasks, for any feasible schedule *K* that reserves additional *k* LO-criticality executions, including the primary or re-executions, there is a feasible schedule *O* that reserves at least *o* LO-criticality executions where $o \geq k$ and the *o* executions are the smallest executions in utilizations.

**Proof:** Suppose that the boundaries of *x* as explained in Figure 1 returned by *K* are $x_{k1}$ and $x_{k2}$ and the ones returned by *O* are $x_{o1}$ and $x_{o2}$. We prove the lemma by showing that in *O* it never uses more room than *K* to reserve any given number of LO-criticality executions, or it always has $x_{o1} \leq x_{k1}$ and $x_{o2} \geq x_{k2}$ when $o = k$.

The following mathematical facts are used in the proof.

**Fact 1:** It is known that for a fraction number $\frac{b}{a}$:

$$\text{if } \frac{b}{a} > 0 \text{ and } \frac{b}{a} < 1.0, \text{ then } \frac{b}{a} < \frac{b+y}{a+y} \text{ for y} > 0 \tag{8}$$

A straightforward derivation from the above fact is that:

$$\text{if } y_1 \geq y_2 > 0, \ \frac{b+y_1}{a+y_1} \geq \frac{b+y_2}{a+y_2} \tag{9}$$

Additionally, if it has another fraction number $\frac{d}{c} > 0$ and $\frac{d}{c} < 1.0$, and $\frac{b}{a} \geq \frac{d}{c}$, let $\frac{b}{a} = \frac{d}{c} + \alpha$ where α is the difference between $\frac{b}{a}$ and $\frac{d}{c}$:

$$\text{it has } \frac{d}{c} = \frac{b - a\alpha}{a}, \text{ and thus } \frac{b+y}{a+y} \geq \frac{d+y}{c+y} \tag{10}$$

Combine the (9) and (10), it is concluded that for two fraction numbers $\frac{b}{a}$ and $\frac{d}{c}$, if $0 < \frac{d}{c} \leq \frac{b}{a} < 1.0$, and two real numbers $y_1 \geq y_2 > 0$:

$$\frac{b+y_1}{a+y_1} \geq \frac{d+y_2}{c+y_2} \tag{11}$$

**Fact 2:** A similar, known fact is that for a fraction number $\frac{b}{a}$:

$$\text{if } \frac{b}{a} > 0 \text{ and } \frac{b}{a} < 1.0, \text{ then } \frac{b}{a} > \frac{b-y}{a-y} \text{ if y} > 0 \tag{12}$$

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

It is true that for fraction numbers $\frac{b}{a}$ and $\frac{d}{c}$, if $0 < \frac{d}{c} \leq \frac{b}{a} < 1.0$, and two real numbers $y_1 \geq y_2 > 0$:

$$\frac{b - y_2}{a - y_2} \geq \frac{d - y_1}{c - y_1} \tag{13}$$

The derivation process of (13) is very similar to the one of (11), and is omitted due to the limit of the length of the paper. Basically, (11) indicates that two fraction numbers, if the first one is not smaller than the second one, each one is added a same number to its numerator and denominator. If the number added to the first one is not smaller than the number added to the second one, after the addition, the first number is not smaller than the second number. Similarly, (13) indicates that two fraction numbers, if the first one is not smaller than the second one, a same number is subtracted from the numerator and denominator of each one. If the subtracted number used on the first number is not larger than the subtracted one used on the second number, after the subtraction, the first number is not smaller than the second number.

We use Mathematical Induction to prove the lemma.

Suppose that there is only one LO-execution selected in solution *K*, which must be a primary execution by the requirements. After all HI-executions are assigned and before the first LO-execution is assigned, the solutions *K* and *O* keep the same task set. Thus:

$$x_{o1} = x_{k1} \tag{14}$$

$$x_{o2} = x_{k2} \tag{15}$$

Then *K* selects a LO-execution with a utilization $u_k$ and *O* selects a LO-execution with a utilization $u_o$. Because in *O* it always selects the smallest one, $u_o \leq u_k$:

$$x_{o1} = \frac{U_{HI}^{LO} + u_o}{1 - U_{LO}^{LO} + u_o} \tag{16}$$

$$x_{k1} = \frac{U_{HI}^{LO} + u_k}{1 - U_{LO}^{LO} + u_k} \tag{17}$$

$$x_{o2} = \frac{1 - U_{HI}^{HI} - u_o}{U_{LO}^{LO} - u_o} \tag{18}$$

$$x_{k2} = \frac{1 - U_{HI}^{HI} - u_k}{U_{LO}^{LO} - u_k} \tag{19}$$

According to the facts of (11) and (13), it has $x_{o1} \leq x_{k1}$ and $x_{o2} \geq x_{k2}$.

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Now assume that after *n* LO-executions are selected by solution *K* and solution *O*, respectively, it is still true that $x_{o1} \le x_{k1}$ and $x_{o2} \ge x_{k2}$. Let $U_k$ and $U_o$ represent the total utilizations of the *n* LO-executions selected by *K* and *O*, respectively:

$$\frac{U_{HI}^{LO} + U_o}{1 - U_{LO}^{LO} + U_o} \le \frac{U_{HI}^{LO} + U_k}{1 - U_{LO}^{LO} + U_k} \tag{20}$$

$$\frac{1 - U_{HI}^{HI} - U_o}{U_{LO}^{LO} - U_o} \ge \frac{1 - U_{HI}^{HI} - U_k}{U_{LO}^{LO} - U_k} \tag{21}$$

We consider the case of *n* + 1 in which *K* selects a next LO-execution with a utilization $u_k$ and *O* selects a next LO-execution with a utilization $u_o$. Since $u_o \le u_k$, by the previous mathematical facts, it is still true that:

$$\frac{U_{HI}^{LO} + U_o + u_o}{1 - U_{LO}^{LO} + U_o + u_o} \le \frac{U_{HI}^{LO} + U_k + u_k}{1 - U_{LO}^{LO} + U_k + u_k} x_{o1} \le x_{k1} \tag{22}$$

$$\frac{1 - U_{HI}^{HI} - U_o - u_o}{U_{LO}^{LO} - U_o - u_o} \ge \frac{1 - U_{HI}^{HI} - U_k - u_k}{U_{LO}^{LO} - U_k - u_k} x_{o2} \ge x_{k2} \tag{23}$$

By the requirements of the *Max Executions* problem, there are three cases of the solutions.

**Case 1**: only HI-criticality tasks are schedulable in the HI-criticality modes. This case is trivial because there is no any LO-criticality task that can be schedulable in the HI-criticality mode.

**Case 2**: only the primary executions of some or all LO-criticality tasks are scheduled along with the HI-criticality tasks in the HI-criticality mode. By lemma 1, it is proved that the Algorithm 1 return the largest number of LO-executions assigned because it always selects the executions with the smallest utilizations.

**Case 3**: some re-executions of LO-criticality tasks are scheduled with the executions scheduled in the previous two cases in the HI-criticality mode. Since the Algorithm 1 is optimal for Case 2, there is no any other solution that can assign the same number of additional LO-executions using less space. Lemma 1 can be used to again to prove that Algorithm 1 is also optimal in this case.

Copyright © 2015, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.