Towards a Fault-tolerant, Scheduling Methodology for Safety-critical Certified Information Systems

Jian Lin
University of Houston-Clear Lake, linjian@uhcl.edu

Follow this and additional works at: https://scholarworks.lib.csusb.edu/jitim
Part of the Other Computer Engineering Commons, and the Technology and Innovation Commons

Recommended Citation
Available at: https://scholarworks.lib.csusb.edu/jitim/vol27/iss3/5

This Article is brought to you for free and open access by CSUSB ScholarWorks. It has been accepted for inclusion in Journal of International Technology and Information Management by an authorized editor of CSUSB ScholarWorks. For more information, please contact scholarworks@csusb.edu.
Towards a Fault-tolerant, Scheduling Methodology for Safety-critical Certified Information Systems

Jian (Denny) Lin  
Email: LinJian@UHCL.EDU  
Department of Management Information Systems  
University of Houston - Clear Lake  
2700 Bay Area Blvd  
Houston, Texas 77058, USA

ABSTRACT

Today, many critical information systems have safety-critical and non-safety-critical functions executed on the same platform in order to reduce design and implementation costs. The set of safety-critical functionality is subject to certification requirements and the rest of the functionality does not need to be certified, or is certified to a lower level. The resulting mixed-criticality systems bring challenges in designing such systems, especially when the critical tasks are required to complete with a timing constraint. This paper studies a problem of scheduling a mixed-criticality system with fault tolerance. A fault-recovery technique called checkpointing is used where a program can go back to a recent checkpoint for re-execution when errors are occurred. A novel schedulability test is derived to ensure that the safety-critical tasks are completed before their deadlines and the theoretical correctness is shown.

KEYWORDS: Safety-critical certification; Mixed-criticality systems; Real-time scheduling; Fault-tolerance.

INTRODUCTION

Modern computing systems can execute multiple applications of different criticality or importance, such as safety-critical and non-safety-critical, on a single platform. Criticality is a designation of the level of assurance against failure needed for a system component. In a mixed-criticality computing system, there are two or more distinct levels of criticality for executions of computing applications. Different standards of identifying levels of criticality have been established in different industries. ASILs (Automotive Safety and Integrity Levels) is a risk classification scheme defined by the ISO 26262 - Functional Safety for Road Vehicles standard.
DALs (Design Assurance Levels), which provides five categories of safety assurance levels, is determined from the safety assessment process and hazard analysis by examining the effects of a failure condition in a software system. SILs (Safety Integrity Levels), specifying a target level of risk reduction, is used as a measurement of performance required for a safety instrumented function (SIF). In the functional safety standards based on the IEC 61508 standard, four SILs are defined.

Systems with safety-critical functionality need to be certified for a permission to operate. The authors in [Baruah et al., 2012] discuss such a case for the design and validation processes of certain Unmanned Aerial Vesicles (UAV’s). The functionalities on board such UAV’s may be classified into two levels of criticality:

- Level 1: the mission-critical functionalities, concerning reconnaissance and surveillance objectives, like capturing images from the ground, transmitting these images to the base station, etc.
- Level 2: flight-critical functionalities, to be performed by the aircraft to ensure its safe operation.

The executions of these two levels of functionalities are controlled by an on-board computer and the tasks are executed continuously. Also, these tasks are real-time tasks that are required to provide responsiveness within a timely constraint or before a deadline. For examples, flight-control tasks are executed every certain time to control an aircraft’s direction, altitude and airspeed in flight. If one of these tasks takes too long to complete, it may cause problems to control the aircraft.

For permission to operate such UAV’s over civilian airspace (e.g., for border surveillance), it is mandatory that its flight-critical functionalities be certified correct by civilian Certification Authorities (CA’s) such as the US Federal Aviation Authority (FAA), which tend to be very conservative concerning the safety requirements. System designers ensure both mission-critical and flight-critical functionalities to be correct but the notion of correctness adopted in validating these functionalities is typically less rigorous than the one used by civilian CA’s. The CA’s may require longer timing budgets reserved for the flight-critical tasks to execute than the ones used by the system designers, in order to ensure the aircraft’s safety. A trade-off can be seen in this process. When the designers determine timing characteristics or timing budgets for running the functional tasks, they estimate the values from extensive experiments. By taking the estimates, all designed functionalities are performed successfully in most of the time but exceptions of executing over deadlines may not be guaranteed to be excluded. The more conservative estimate by the CA’s can exclude missing execution deadlines to the greatest extent possible but it may cause a shortage of CPU time resource to
accommodate all of the flight-critical and mission-critical tasks onto the single system. Recently, how to overcome this conflict has become an increasing research trend [Burs and Davis, 2018].

In executing computing tasks, faults or errors may happen during the process which can either produce incorrect results or cause real-time tasks to miss their deadlines. Permanent and transient faults are the two categories of errors that happen the most frequently. Permanent faults, such as hardware damage and shutdown, cannot be recovered. Transient faults, by contrast, can be recovered by re-executing the faulty task. 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 systems transient faults occur much more frequently than permanent faults do [Castillo et al., 1982; Iyer and Rossetti, 1986]. Generally, there are two major techniques to recover transient faults, primary-backup execution [Al-Omari et al., 2004] and checkpointing [Punnekkat et al., 2001]. A backup is an exact copy of an execution of a task. A checkpoint is a regularly-saved state of a task, which consists of values of data variables and contents of system registers. An acceptance test that ensures the program’s successful execution must be run before saving the necessary data. In the primary-backup execution technique, the whole faulty task is re-executed where in the checkpointing technique a re-execution of the affected task is performed from the most recent checkpoint.

In this work, we solve the certification problem in mixed-criticality systems from a perspective of scheduling. We work on a methodology that focuses on executions of mixed-criticality, real-time tasks and fault tolerance, particularly in using the technique of checkpointing. To the best of the author's knowledge, this is the first work that considers using checkpointing in scheduling mixed-criticality tasks. The rest of the paper is organized as follows. Next section discusses some preliminary and related works. Section 3 formally introduces the system model and problem definition. Then, a novel schedulability test condition for a set of mixed-criticality tasks with fault tolerance is derived. We also present an example to explain how to use the test condition. The last section summarizes and concludes the work.

**RELATED WORKS**

Different task models have been built to characterize an execution of a real-time task. In a periodic task, job instances arrive regularly with a fixed inter-interval. A job instance of a periodic task in general is required to complete before an arrival of the next instance. Tasks with irregular arrival intervals are called aperiodic tasks.
Aperiodic tasks that have a minimum inter-arrival time are called sporadic tasks. The first real-time scheduling paper was published in [Liu and Layland, 1973]. Since then, a tremendous number of works have been done in the field. In primarily, there are two types of real-time scheduling algorithms, static-priority and dynamic-priority. In a scheduling process, tasks are assigned priorities which are used to determine their order in execution. In a static-priority algorithm, priorities are assigned off-line and do not change during run-time. In contrast, a dynamic-priority algorithm schedules the tasks based on their priorities assigned on-line. For examples, Earliest Deadline First (EDF) is a classical, dynamic-priority algorithm that always selects a task closest to its deadline to run. Rate Monotonic (RM) is a static-priority algorithm that assigns priorities to periodic tasks based on the lengths of their periods. Since a length of a period of a task does not change, the priority stays the same during the task's execution. In practice, static-priority algorithms are simpler to implement in an operating system and dynamic-priority algorithms are more complex to predict the scheduling outcomes. However, dynamic-priority algorithms generally have a better utilization of CPU time. For further information about real-time scheduling, please refer to the following texts [Cheng, 2002; Liu, 2000; Krishna and Shin, 1997].

In the past several years, mixed-criticality systems became a very popular research topic in designing critical information systems. Computing tasks with different criticality sharing the same resource on a single hardware platform can reduce design and implementation costs. However, as we mentioned earlier, it also brings challenges to confirm the schedulability of these tasks. It is well-known that conventional scheduling methods cannot satisfactorily address these challenges and the mathematical intractability of solving these problems has been proved in [Baruah et al., 2012]. In the existing works such as those in [De Niz et al., 2009; Lakshmanan et al., 2010; Baruah and Vestal, 2008; Ekberg and Yi, 2012; Guan et al., 2011; Baruah et al., 2008; Baruah et al., 2010], tasks running on a mixed-criticality system are classified into two categories, safety-critical or HI-criticality, and non-safety-critical or LO-criticality. A HI-criticality task may have two estimated execution times, one from the CA's certification, and another from the system designers. At the beginning, both LO-criticality and HI-criticality tasks are scheduled by using their shorter estimated timing budgets. Once a HI-criticality task uses out its timing budget without a completion, it signals that the execution times estimated by the system designers are not trustworthy. At this moment, all HI-criticality tasks are assumed to run with their longer execution times required by the CA’s. Simultaneously, all LO-criticality tasks are dropped in order to keep the safety of executing those HI-criticality tasks successfully.
Mixed-criticality systems with fault tolerance are also explored in the research community. In [Pathan 2014], the authors design a schedulability test for using the primary-backup technique. In [Huang et al., 2014], the authors describe a method to convert the fault-tolerant problem into a standard scheduling problem in a mixed-criticality system. In one of our earlier works, the EDF scheduling algorithm and the primary-backup technique are used to maximize the number of scheduled LO-criticality tasks while all of the HI-criticality tasks are schedulable [Lin et al., 2015].

At the time of writing this paper, none of existing works has engaged in solving the problem of using the checkpointing technique.

SYSTEM MODEL, PROBLEM DEFINITION AND SCHEDULABILITY TEST

System Model and Problem Definition

We consider that a mixed-criticality system consists of a set of $N$ sporadic tasks $T = \{T_1, T_2, ..., T_N\}$ where consecutive instances of a task $T_i$ arrive with a minimum inter-interval, denoted by $P_i$. In order to ensure the schedulability in the worst-case scenario, we assume that the instances of each task arrive with their maximum frequency. In other words, each task has an instance to complete for every $P_i$ which is called a period of $T_i$. For each task, the value of the worst-case execution time (WCET) is significant due to the requirement of having no deadline violations. The time between each task instance’s arrival and its deadline is called a relative deadline. A relative deadline of $T_i$ is denoted as $D_i$ where $D_i = P_i$. There are two criticality levels in the system, LO or HI. A task is either a LO-criticality or a HI-criticality task and its criticality is denoted by $X_i$, $X_i \in \{LO, HI\}$. For a HI-criticality task, it has two WCETs as $C_i(LO)$ and $C_i(HI)$ and a LO-criticality task may have a $C_i(LO)$ only. It is assumed that $C_i(HI) \geq C_i(LO)$. When the system starts, all tasks may have an infinite sequence of instances to execute. Initially, all HI-criticality and LO-criticality tasks are scheduled using their $C(LO)$s 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 $C(HI)$s immediately. The system is thus switched into a HI-criticality mode. All of the LO-criticality tasks are dropped from the execution in order to maintain the feasibility of executing the HI-criticality tasks.

We also define the faults arrival pattern that is used in our analysis. There is no difficulty to understand that there is no solution that can accommodate unlimited
errors. In this work, we assume that there is a minimum inter-interval of $P_f$ between any two faults' arrival. The faults considered are transient faults which can be recovered by re-executing the faulty task. Checkpoints are used in recovering the faulty tasks from errors. A HI-criticality task may be checkpointed into $m_i(LO)$ segments in its $C_i(LO)$ and $m_i(HI)$ segments in its $C_i(HI)$, where $m_i(HI) \geq m_i(LO)$. The interval of each segment in the same task, denoted by $I_i$, is assumed to be the same except of the last segment (a WCET may not be divisible by an $I_i$). Also, we assume that there is no error happened during a creation of a checkpoint and an acceptance test.

The problem we target to solve is formally defined as follows. Given a task-set of $T$, each task is defined as $T_i = \{P_i, D_i, r_i, X_i, C_i(LO), C_i(HI), m_i(LO), m_i(HI), I_i\}$ in which $r_i$ is a unique integer that indicates a static priority of $T_i$. The smaller the integer, the higher priority it indicates. The tasks are scheduled using each task's static priority. Assuming that faults arrive between a minimum interval of $P_f$, determine the task-set's schedulability that all tasks are schedulable in a LO-criticality mode and all HI-criticality tasks are schedulable when the system is switched to and in a HI-criticality mode.

**Schedulability Test**

**Scheduling without Fault Tolerance**

In real-time scheduling, a standard response-time analysis is used to determine schedulability of a set of tasks using static priorities [Joseph and Pandya, 1986]. In a response-time analysis, each task's worst-case response time is calculated. A response time is defined as the time between a task’s arrival and its completion. If the worst-case response time of a task is smaller than or equal to the task's relative deadline, the task is schedulable. When calculating a task's response time, only the tasks with higher priority have impacts to it. The response time value $R_i$ is obtained from the following formula (where $C$ denotes the WCET and $hp_i$ denotes the set of tasks with priority higher than that of task $T_i$):

$$R_i = C_i + \sum_{j \in hp_i} \left( \left\lceil \frac{R_i}{P_j} \right\rceil \times C_j \right)$$

(1)

This is solved using standard techniques for solving recurrence relations. The recurrence calculation stops when $R_i$ on both sides are equal. To determine a task set's schedulability, it can be done by calculating all tasks' response times in the set.
In [Baruah et al., 2011], the authors define three conditions that need to be satisfied in order to decide the schedulability for a mixed-criticality system:

i. All tasks' response times are not larger than their relative deadlines by using their $C(LO)$.

ii. All $HI$-criticality tasks' response times are not larger than their relative deadlines by using their $C(HI)$.

iii. No $HI$-criticality tasks miss their deadlines during a switch from a $LO$-criticality mode to a $HI$-criticality mode.

In practice, it is possible that conditions i and ii are satisfied and condition iii is failed. This is because when a system switches its mode, some of the $LO$-criticality tasks may have been executed for a certain amount of time. As a result, it may cause some $HI$-criticality tasks to miss deadlines due to a lack of enough CPU time for the execution of $C(HI)$ before their deadlines. We explain such a failure possibility by considering an example of a task-set as in Table 1.

### Table 1. Example of a set of three mixed-criticality tasks

<table>
<thead>
<tr>
<th>$T_i$</th>
<th>$X_i$</th>
<th>$r_i$</th>
<th>$P_i$</th>
<th>$D_i$</th>
<th>$C_i(LO)$</th>
<th>$C_i(HI)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$LO$</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>$T_2$</td>
<td>$HI$</td>
<td>2</td>
<td>6</td>
<td>6</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$T_3$</td>
<td>$HI$</td>
<td>3</td>
<td>10</td>
<td>10</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

By verifying the schedulability of the $LO$-criticality mode, it can replace the $C_i$ in (1) by $C_i(LO)$. That is:

$$R_i^{LO} = C_i(LO) + \sum_{\forall j \in hp_i} \left( \left\lfloor \frac{R_i^{LO}}{P_j} \right\rfloor \times C_j(LO) \right)$$  \hspace{1cm} (2)

Similarly, by verifying the schedulability of the $HI$-criticality mode, it can replace the $C_i$ in (1) by $C_i(HI)$ and exclude all $LO$-criticality tasks ($hp_iH$ denotes the set of $HI$-criticality tasks with priority higher than that of task $T_i$).

$$R_i^{HI} = C_i(HI) + \sum_{\forall j \in hp_iH} \left( \left\lfloor \frac{R_i^{HI}}{P_j} \right\rfloor \times C_j(HI) \right)$$  \hspace{1cm} (3)

By using (2) and (3), the following can be obtained:

$R_1^{LO} = 2$, $R_2^{LO} = 4$ and $R_3^{LO} = 10$;

$R_2^{HI} = 3$ and $R_3^{HI} = 6$
Both conditions i and ii are satisfied. However, the schedule in Figure 1 shows that condition iii is violated. At time instant 10, $T_3$ has used 2 time units as its $C(LO)$ without a completion. It signals the system and the system is switched to a HI-criticality mode. $T_1$ is dropped and both $T_2$ and $T_3$ increase their WCETs to 3 immediately. It can be seen that $T_3$ misses its deadline during the mode switching.

In [Baruah et al., 2011], it is shown that verifying the schedulability for condition iii is unlikely to be tractable in that all release patterns of all sporadic tasks would need to be tested. A sufficient but not necessary condition is proposed in the work (The response time used in condition iii is denoted as $R_i^*$):

$$R_i^* = C_i(HI) + \sum_{j \in h_p_i \mid H} \left[ \frac{R_i^L}{P_j} \right] \times C_j(HI)$$

$$+ \sum_{k \in h_p_i \mid L} \left[ \frac{R_i^{LO}}{P_k} \right] \times C_k(LO)$$

The equation (4) not only counts the computation impact from the HI-criticality tasks with higher priority than the one of $T_i$, it also "caps" the interference from the LO-criticality tasks (the set of $h_p_i \mid L$) because a mode switching must happen before $R_i^{LO}$.

**Figure 1. A Schedule of three mixed-criticality task**
Scheduling with Checkpoints

We extend the work described in section 3.2.1 to recover faults by using checkpoints. Checkpoints separate an execution of a task into segments. It reduces the time required for a re-execution for errors, up to the length of each segment’s interval. By using checkpoints, additional overhead has to be considered and it is not trivial [Punnekkat et al., 2001]. Before a checkpoint is created, an acceptance needs to be performed to ensure the result of the execution in the current segment to be correct. Then, the variable states and registers values are saved before it starts an execution for the next segment. We use $O$ to denote the overhead of one acceptance test and one saving of the program states. For a WCET with $m$ segments, the total overhead is $m \times O$. This is from $m - 1$ times of creating the checkpoints plus one time of saving states at the beginning and one time of acceptance test at the final completion. When errors are detected in an acceptance test, it will bring an additional $I + O$ time units to the execution time. The $I$ is the segment interval for a re-execution and the $O$ is for another time of saving states and acceptance test. Please note that we assume that two consecutive faults arrive with at least a separation of $P_f$ time units.

To verify the LO-criticality schedulability with checkpoints:

$$R_{LO}^i = C_i(LO) + O_i \times m_i(LO)$$

$$+ \sum_{j \in hp_i} \left\lfloor \frac{R_{LO}^i}{P_j} \right\rfloor \times (C_j(LO) + O_j \times m_j(LO))$$

$$+ \left\lfloor \frac{R_{LO}^i}{P_f} \right\rfloor \max_{k \in hp_i \cup \{i\}} (O_k + I_k)$$

(5)

The sum consists of three inclusions for the response time: $T_i$'s computation time and checkpointing overhead, all higher-priority tasks' computation times and checkpointing overhead and the maximum re-execution time of the number of faults that can occur within $R_{LO}^i$. 

©International Information Management Association, Inc. 2017
ISSN: 1941-0479-On-line Copy
Similarly, the following is derived to verify the HI-criticality schedulability with checkpoints:

\[
\begin{align*}
R_i^{HI} &= C_i(HI) + O_i \times m_i(HI) \\
&+ \sum_{\forall j \in hp_i H} \left[ \frac{R_i^{HI}}{P_j} \right] \times (C_j(HI) + O_j \times m_j(HI)) \\
&+ \left[ \frac{R_i^{HI}}{P_f} \right] \max_{\forall k \in hp_i H \cup \{i\}} (O_k + I_k)
\end{align*}
\]

(6)

In order to verify the schedulability during a mode switching, we need to include the checkpointing overhead and total re-execution time for the faults from the HI-criticality tasks and from the LO-criticality tasks before the switching.

\[
\begin{align*}
R_i^* &= C_i(HI) + O_i \times m_i(HI) \\
&+ \sum_{\forall j \in hp_i H} \left[ \frac{R_i^*}{P_j} \right] \times (C_j(HI) + O_j \times m_j(HI)) \\
&+ \left[ \frac{R_i^*}{P_f} \right] \max_{\forall k \in hp_i H \cup \{i\}} (O_k + I_k) \\
&+ \sum_{\forall q \in hp_i L} \left[ \frac{R_i^{LO}}{P_q} \right] \times (C_q(LO) + O_q \times m_q(LO)) \\
&+ \left[ \frac{R_i^{LO}}{P_f} \right] \max_{\forall s \in hp_i L \cup \{i\}} (O_s + I_s)
\end{align*}
\]

(7)

By a more thorough analysis, it can be seen that there is an overlap in (7) between \(R_i^*\) and \(R_i^{LO}\). Because \(R_i^*\) includes \(R_i^{LO}\) and \(R_i^*\) must be greater than \(R_i^{LO}\), the number of faults that may occur by \(R_i^{LO}\) counts two times. In fact, by \(R_i^{LO}\), errors may occur in any task with higher priority, and after \(R_i^{LO}\) we only need to count the faults from the HI-criticality tasks with higher priority. Thus, (7) can be revised and improved as:
Towards a Fault–tolerant, Scheduling Methodology for Safety-critical Certified Information Systems

J. Lin

© International Information Management Association, Inc. 2017
ISSN: 1941-6679

\[
R_i^* = C_i(HI) + O_i \times m_i(HI)
\]

\[
+ \sum_{\forall j \in h_p H} \left( \left\lfloor \frac{R_i^*}{P_{j}} \right\rfloor \times C_j(HI) + O_j \times m_j(HI) \right)
\]

\[
+ \left[ \frac{R_i^* - R_i^{LO}}{P_f} \right] \max_{\forall k \in h_p H \cup \{i\}} (O_k + I_k)
\]

\[
+ \sum_{\forall q \in h_p L} \left( \left\lceil \frac{R_i^{LO}}{P_q} \right\rceil \times C_q(LO) + O_q \times m_q(LO) \right)
\]

\[
+ \left[ \frac{R_i^{LO}}{P_f} \right] \max_{\forall s \in h_p L \cup \{i\}} (O_s + I_s)
\]

The equations (5), (6) and (8) can be also used to determine the schedulability for using the primary-backup technique because the primary-backup technique is a special case of the checkpointing technique. In the primary-backup technique, the number of segments can be taken as 1 and the additional overhead for saving states and acceptance test is just one time of \(O\). Each re-execution after errors are detected is equal to the length of the faulty task's WCET. There is another one time of \(O\) associated with each time of the re-execution. Thus, the equations (5), (6) and (8) can be modified to be using with the primary-backup technique as follows.

\[
R_i^{LO} = C_i(LO) + O_i
\]

\[
+ \sum_{\forall j \in h_p_i} \left( \left\lfloor \frac{R_i^{LO}}{P_{j}} \right\rfloor \times (C_j(LO) + O_j) \right)
\]

\[
+ \left[ \frac{R_i^{LO}}{P_f} \right] \max_{\forall k \in h_p_i \cup \{i\}} (C_k(LO) + O_k)
\]

\[
R_i^{HI} = C_i(HI) + O_i
\]

\[
+ \sum_{\forall j \in h_p H} \left( \left\lfloor \frac{R_i^{HI}}{P_{j}} \right\rfloor \times (C_j(HI) + O_j) \right)
\]

\[
+ \left[ \frac{R_i^{HI}}{P_f} \right] \max_{\forall k \in h_p H \cup \{i\}} ((C_j(HI) + O_k)
\]
\[ R_i^* = C_i(HI) + O_i \]
\[ + \sum_{\forall j \in h_p_i} \left[ \frac{R_j^*}{P_j} \right] \times (C_j(HI) + O_j) \]
\[ + \left[ \frac{R_i^* - R_i^{LO}}{P_f} \right] \max_{\forall k \in h_p_i} (C_k(HI) + O_k) \]
\[ + \sum_{\forall q \in h_p_i} \left[ \frac{R_q^{LO}}{P_q} \right] \times (C_q(LO) + O_q) \]
\[ + \left[ \frac{R_i^{LO}}{P_f} \right] \max_{\forall s \in h_p_i} (C_s(HI) + O_s) \]  

(11)

A Demonstrative Example
We demonstrate an example to show how to use the schedulability test condition we derived in this paper. Consider another task-set with three mixed-criticality tasks in Table 2. We show how to calculate the response-time for \( T_3 \).

Table 2. Example of a set of three mixed-criticality tasks with using checkpoints, \( Pf = 20 \).

<table>
<thead>
<tr>
<th>( T_i )</th>
<th>( X_i )</th>
<th>( r_i )</th>
<th>( P_i )</th>
<th>( D_i )</th>
<th>( C_i(LO) )</th>
<th>( C_i(HI) )</th>
<th>( m_i(LO) )</th>
<th>( m_i(HI) )</th>
<th>( O_i )</th>
<th>( I_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_1 )</td>
<td>( LO )</td>
<td>1</td>
<td>100</td>
<td>100</td>
<td>15</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( T_2 )</td>
<td>( HI )</td>
<td>2</td>
<td>120</td>
<td>120</td>
<td>10</td>
<td>15</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>( T_3 )</td>
<td>( HI )</td>
<td>3</td>
<td>140</td>
<td>140</td>
<td>25</td>
<td>40</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

The following calculation is to determine whether or not \( T_3 \) is schedulable in the LO-criticality mode (verifying condition i).

\( R_3^{LO} = 25 + 5 + 15 + 3 + 10 + 2 = 60 \)
\( R_3^{LO} = 25 + 5 + \left[ \frac{60}{100} \right] \times 18 + + \left[ \frac{60}{120} \right] \times 12 + + \left[ \frac{60}{20} \right] \times 6 = 78 \)
\( R_3^{LO} = 25 + 5 + \left[ \frac{78}{100} \right] \times 18 + + \left[ \frac{78}{120} \right] \times 12 + + \left[ \frac{78}{20} \right] \times 6 = 84 \)
\( R_3^{LO} = 25 + 5 + \left[ \frac{84}{100} \right] \times 18 + + \left[ \frac{84}{120} \right] \times 12 + + \left[ \frac{84}{20} \right] \times 6 = 90 \)
\( R_3^{LO} = 25 + 5 + \left[ \frac{90}{100} \right] \times 18 + + \left[ \frac{90}{120} \right] \times 12 + + \left[ \frac{90}{20} \right] \times 6 = 90 \)
\( R_3^{LO} = 90 \)
Towards a Fault–tolerant, Scheduling Methodology for Safety-critical Certified Information Systems

J. Lin

© International Information Management Association, Inc. 2017
ISSN: 1941-6679

On-line Copy

The calculation above is explained as follows. Because \( T_1 \)'s and \( T_2 \)'s priorities are higher than \( T_3 \)'s, \( T_1 \) and \( T_2 \) are executed before \( T_3 \) when all of them start at time instant 0. The calculation starts by adding all of the three tasks' \( C(LO) \)s and the time overhead used to create the checkpoints, which is equal to 60. The recurrence calculation continues by including the higher priority tasks' \( C(LO) \)s, checkpointing costs and the execution time for the maximum number of re-execution segments, until at time instant 90 the total execution demand does not increase (both sides are equal). According to the formula (5), \( R_3^{LO} = 90 \). \( R_3^{HI} \) is calculated similarly by considering \( HI \)-criticality tasks only as defined in the formula (6).

\[
R_3^{HI} = 40 + 8 + 15 + 3 = 66
\]
\[
R_3^{HI} = 40 + 8 + \frac{66}{120} \times 18 + \frac{66}{20} \times 6 = 90
\]
\[
R_3^{HI} = 40 + 8 + \frac{90}{120} \times 18 + \frac{90}{20} \times 6 = 96
\]
\[
R_3^{HI} = 40 + 8 + \frac{96}{120} \times 18 + \frac{96}{20} \times 6 = 96
\]
\[
R_3^{HI} = 96
\]

Since \( R_3^* \) must be greater than \( R_3^{LO} \) and \( R_3^{HI} \), \( R_3^* \) is initialized to be the greater one between \( R_3^{LO} \) and \( R_3^{HI} \), so \( R_3^* \) starts at 96. The following shows the calculation process of \( R_3^* \) based on the formula (8).

\[
R_3^* = 40 + 8 + \frac{96}{120} \times 18 + \frac{96-90}{20} \times 6 + \frac{90}{100} \times 18 + \frac{90}{20} \times 6 = 114
\]
\[
R_3^* = 40 + 8 + \frac{114}{120} \times 18 + \frac{114-90}{20} \times 6 + \frac{90}{100} \times 18 + \frac{90}{20} \times 6 = 120
\]
\[
R_3^* = 40 + 8 + \frac{120}{120} \times 18 + \frac{120-90}{20} \times 6 + \frac{90}{100} \times 18 + \frac{90}{20} \times 6 = 120
\]
\[
R_3^* = 120
\]

By the above calculation, all of the calculated \( R_3^{LO} \), \( R_3^{HI} \) and \( R_3^* \) are not larger than \( T_3 \)'s relative deadline \( D_3 \) which is 140, so \( T_3 \) is schedulable. The schedulability tests of \( T_1 \) and \( T_2 \) use the same technique and we omit the details for the sake of avoiding a lengthy paragraph. In fact, both \( T_1 \) and \( T_2 \) are schedulable and hence the task-set is schedulable.

**SUMMARY AND FUTURE WORKS**

Checkpointing is a widely used technique for fault-tolerant computing. This paper solves the problem of applying checkpointing for scheduling mixed-criticality
tasks. A new sufficient schedulability test condition is derived and its theoretical correctness is shown along with the derivation.

In the example shown in section 3.2.3, it is apparently that $T_3$ is not schedulable by using the primary-backup technique. This is because when $P_f = 20$, errors can occur in every execution of $T_3$. In the worst case there are two errors occurred in every instance of $T_3$. Considering that every time $T_3$ needs to restart the whole execution for an error, a $T_3$ instance will never complete its execution by its deadline. It is seen that for tasks un-schedulable upon using a complete re-execution, it is possible to make the tasks schedulable by using checkpoints. Our future works include optimization techniques for the placement of checkpoints in scheduling mixed-criticality tasks.

REFERENCES


