2013

An Efficient Fault Recovery Algorithm in Multiprocessor Mixed-Criticality Systems

Guangdong Liu
University of Nebraska-Lincoln, gliu@cse.unl.edu

Ying Lu
University of Nebraska-Lincoln, ying@unl.edu

Shige Wang
General Motor Global Research and Development, shige.wang@gm.com

2013 IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing
Pages: 2006 - 2013, DOI: 10.1109/HPCC.and.EUC.2013.289

An Efficient Fault Recovery Algorithm in Multiprocessor Mixed-Criticality Systems

Guangdong Liu, Ying Lu
Department of Computer Science and Engineering
University of Nebraska-Lincoln
Lincoln, NE, USA
{gliu, ylu}@cse.unl.edu

Shige Wang
Electrical and Controls Integration Lab
General Motor Global Research and Development
Warren, MI, USA
shige.wang@gm.com

***Abstract*—Recent years, there is an increasing interest of integrating mixed-criticality functionalities onto a shared computing platform in automotive, avionics and the control industry. The benefits of such an integration include reduced hardware cost and high computational performance. Also, new challenges appear as a result of the integration since interferences across tasks with different criticalities are introduced and these interferences could potentially lead to catastrophic results. Failures are likely to be more frequent due to the interferences. Hence, it is becoming increasingly important to deal with faults in mixed-criticality systems. Although several approaches have been proposed to handle failures in mixed-criticality systems, they come either with a high cost due to a hardware replication (spatial redundancy) or with a poor utilization due to re-execution (time redundancy).**

In this paper, we study a scheme that provides fault recovery through task reallocations in response to permanent faults in multiprocessor mixed-criticality systems. We present an algorithm to minimize the number of task reallocations while retaining the promise that the most critical applications continue to meet their deadlines. The performance evaluation of the proposed algorithm is carried out by comparing it with two baseline algorithms. In order to evaluate the performance of algorithms from the perspective of mixed-criticality systems, we choose the state of art metric called ductility to formally measure the effects of deadline misses for tasks with different criticality levels. Under this metric, a high-criticality task is considered more important than all low-criticality tasks combined. The simulation results confirm the effectiveness of our proposed algorithm in both minimizing the number of task reallocations and retaining the promised performance of high-criticality tasks.

I. Introduction

Modern large embedded real-time systems, such as those in avionics, automotive and robotics applications, typically comprise many diverse functions with different criticality levels. Traditional approaches implement the system using an independent architecture. In such an architecture, the functions have separate, dedicated devices for their software execution. With each control functionality running on its own computer system, all associated cost of acquisition, space, power, weight, cooling, installation, and maintenance increases. In addition, the lack of properly integrated control caused by the artificial separation of functions is one of the most common root causes for many design, integration, quality, and performance problems [9].

To address the challenges of integrating more intelligent, complex and cooperating controls in advanced embedded real-time systems, the integrated architecture is proposed and advocated. The key characteristic of integrated architecture is that functionalities of different criticalities are consolidated onto a shared computing platform. However, such sharing also introduces interferences across tasks which could potentially lead to catastrophic results [8]. Failures are likely to be more frequent due to the interferences across tasks in mixed-criticality systems. Therefore, schemes for handling faults in mixed-criticality systems become increasingly important.

Several approaches have been proposed for handling failures in mixed-criticality systems. Saraswat et al. proposed a task migration scheme for handling permanent faults and check-pointing with rollback recovery for transient faults [10, 11]. Aysan et al. presented a generic fault-tolerant scheduling framework for mixed-criticality systems [2, 14]. Four criticality levels, including non-critical, critical, highly-critical and ultra-critical, are considered. A feasibility window is derived to reserve necessary slacks for the re-execution of critical, highly-critical and ultra-critical task instances while a best-effort service is provided to non-critical task instances. Sebastian et al. presented a reliability analysis for mixed-criticality systems when using redundancy in time and space dimensions to detect and correct errors [1].

These solutions, however, come either with a high cost due to a hardware replication (spatial redundancy) or with a poor utilization due to re-execution (time redundancy). Moreover, most of them are designed with reservation-based scheduling, which according to Li and Baruah [7] is inferior to priority-based scheduling in mixed-criticality systems.

In priority-based task scheduling for mixed-criticality systems, special attention is needed to avoid criticality inversion. Criticality inversion occurs when a high-criticality task misses its deadline due to interferences from low-criticality tasks [8]. In order to eliminate criticality inversion problem and improve processor utilization in mixed-criticality systems, [8] proposed a scheduling scheme called zero-slack scheduling in which both priorities and criticalities of tasks are considered when deciding the task execution order. In [6], Lakshmanan et al. generalized the zero-slack scheduling to a multiprocessor system. However, neither of these schemes take faults into account when designing the algorithms.

In this paper, we study a scheme that provides efficient fault recovery through task reallocations in response to permanent node failures in multiprocessor mixed-criticality systems which adopt priority-based scheduling. We present

978-0-7695-5088-6/13 $31.00 © 2013 IEEE
DOI 10.1109/HPCC.and.EUC.2013.289

an algorithm to minimize the number of task reallocations while retaining the promise that the most critical applications continue to meet their deadlines. In order to evaluate the performance of algorithms from the perspective of mixed-criticality systems, we choose the state of art metric called ductility [6] to formally measure the effects of deadline misses for tasks with different criticality levels. Under this metric, a high-criticality task is considered more important than all low-criticality tasks combined.

The rest of this paper is organized as follows. Section II presents the background of zero-slack scheduling and the metric we adopt to evaluate the performance of mixed-criticality systems. Section III introduces the system model and characterizes the problem. Section IV describes the task reallocation algorithms. In Section V the performance of the task reallocation algorithms are analyzed and discussed. Section VI briefly presents related work in mixed-criticality systems. Finally, in Section VII we conclude the paper.

II. Background

A. Zero-Slack Scheduling in Uniprocessor

In [8], de Niz el al. proposed a new real-time scheduling scheme called zero-slack scheduling in which both priorities and criticalities of tasks are considered when deciding the task execution order. Traditionally, priorities are assigned to real-time tasks based on two major considerations. One is to maximize the schedulable utilization. The other is to meet the deadlines of all tasks in the system. However, these priority-based scheduling schemes do not consider criticality at all. In other words, they are agnostic to the fact that applications with different criticality levels could be integrated onto a shared computing platform. Therefore, it is possible that a low-criticality task may have a high priority and may cause a high-criticality task with a low priority to miss its deadline. This problem is identified and called the criticality inversion problem [8]. In order to solve this problem, both priorities and criticalities should be considered when scheduling tasks.

A new task model introduced in [8] is used to describe the mixed-criticality system. In the model, there are two worst-case execution times for a task, corresponding to two scenarios respectively. Formally, a task τ_i is defined as:

$$\tau_i = (C_i, C_i^o, T_i, D_i, \varepsilon_i) \tag{1}$$

where: (a) C_i is its worst-case execution time under non-overload conditions, (b) C_i^o is the overload execution budget, (c) T_i is the period of the task, (d) D_i is the task deadline (with $D_i \leq T_i$), and (e) ε_i is the criticality of the task and the lower the value, the higher the criticality. The priority of a task is assigned based on the scheduling strategy adopted by the mixed-criticality system. For instance, if rate monotonic scheduling is used, a task with a shorter period is given a higher priority.

In the zero-slack scheduling algorithm developed in [8], each task can run in two execution modes. One is called normal mode (N mode) in which tasks are scheduled based on their priorities in order to obtain a high schedulable utilization. The other is called critical mode (C mode) in which tasks are scheduled based on their criticalities. In C mode, a high-criticality task can suspend lower-criticality tasks in order to meet its deadline. Initially, each task runs in N mode. When a task's actual execution exceeds a certain time instant, it will switch to C mode in which lower-criticality tasks are suspended in order to meet this high-criticality task's deadline. This time instant is identified as the zero-slack instant because it leaves no slack in C mode after the completion of C_i^o. It is the last time instant when lower-criticality tasks can free enough time to a high-criticality task to finish before its deadline. An algorithm in calculating the zero-slack instant is given in [8]. The zero-slack scheduling scheme can be built on top of any traditional priority-based preemptive scheduling. In [8], de Niz et al. integrated zero-slack scheduling with the rate monotonic scheduling and evaluated the Zero-Slack Rate Monotonic (ZSRM) algorithm's performance. It has been proven that the zero slack scheduling ensures the desired *asymmetric protection* property for mixed criticality systems, by only preventing interference to high-criticality tasks from lower-criticality tasks [8].

B. Ductility of Mixed-Criticality Systems

In [6], Lakshmanan et al. generalized the zero-slack scheduling to a multiprocessor system. In order to capture the mixed-criticality property, they derived a new metric called ductility matrix to formally evaluate a scheduling algorithm's performance from the perspective of mixed-criticality systems. The ductility matrix describes the system's timing behavior under all possible workload scenarios. A workload scenario is defined by whether tasks in a certain criticality level are in the overload operating state or not, i.e., characterized by a workload vector $< W_1, W_2, \cdots, W_k >$, where W_i is an indicator variable that denotes the operating state of tasks with criticality value i: $W_i = 0$ means that all tasks with criticality i are in the normal operating state and $W_i = 1$ denotes that at least one task with criticality i is in the overload operating state. With k criticality levels, there are 2^k different workload scenarios since tasks in each criticality level can be in either overload or normal operating state. In addition, a scalar equivalent known as system workload (w) is computed from the workload vector as: $w = \sum_{g=1}^{k}(2^{k-g}W_g)$. The ductility matrix $D = (d_{r,c})$ describes whether all tasks with a criticality level c can meet their deadlines under system workload $w = 2^k - r$:

- when $d_{r,c} = 1$, it means that all tasks in criticality level c can meet their deadlines under workload $w = 2^k - r$.
- when $d_{r,c} = 0$, it means that at least one task in criticality level c cannot meet its deadline under workload $w = 2^k - r$.

In order to simplify the evaluation of different scheduling algorithms, Lakshmanan et al. further defined a scalar equivalent of the ductility matrix that can be ordered based on magnitudes. To obtain the scalar, called normalized ductility and ν value (normalized to the range [0, 1]) [6], the following formulas are used:

$$\nu = \frac{P_d(D)}{1 - \frac{1}{2^k}} \quad (2)$$

$$\text{where } P_d(D) = \sum_{c=1}^{k} \{ \frac{1}{2^c} \frac{\sum_{r=1}^{2^k} d_{r,c}}{2^k} \} \quad (3)$$

The projection function $P_d(D)$ treats a task in a high-criticality level as absolutely more important than any lower-criticality tasks and thus it is always more important to meet the deadline of a high-criticality task than to meet deadlines of any lower-criticality tasks combined. The normalized ductility enables researchers to evaluate a scheduling algorithm from the perspective of mixed-criticality systems. The higher the ν value, the more ductility and the better performance a mixed-criticality system has.

III. System Model and Problem Formulation

We assume that a mixed-criticality task set consists of n independent periodic tasks: $T = \{\tau_1, \tau_2, ..., \tau_n\}$. We adopt the same task model as that in [8], i.e., a task τ_i is specified by a tuple $(C_i, C_i^o, T_i, D_i, \varepsilon_i)$ (Equation (1)). A multiprocessor mixed-criticality system is modeled as a set of homogeneous interconnected processors: $N = \{P_1, P_2, ..., P_m\}$.

Due to the proven better performance of *Compress-on-Overload Packing* (COP) algorithm [6] in comparison to best-fit decreasing, worst fit-decreasing and first-fit decreasing algorithms, initially tasks are assumed to be partitioned to the multiprocessor mixed-criticality system following COP algorithm. It is a two-phase task allocation algorithm. The basic idea is to allocate high-criticality tasks first and then low-criticality tasks. In the first phase, tasks are sorted in decreasing order of criticality level and a tie is broken by following decreasing order of task overload utilization $\frac{C_i^o}{T_i}$. A processor is considered to have enough space for a task if the overloaded fullness level $\sum_{\forall i} \frac{C_i^o}{T_i}$ does not exceed the schedulable utilization bound. After the first phase, all remaining tasks will be sorted again in decreasing order of criticality and a tie is broken by following decreasing order of normal utilization $\frac{C_i}{T_i}$. In this phase, a processor is considered to have enough space for a task if the normal fullness level $\sum_{\forall i} \frac{C_i}{T_i}$ does not exceed the threshold. We also assume that Zero-Slack Rate Monotonic (ZSRM) scheduling [8] is used to schedule tasks in a processor.

We use the normalized ductility (i.e., ν value presented in Equation (2)) to evaluate the performance of a multiprocessor mixed-criticality system. ν value is the state-of-art metric in evaluating scheduling algorithms of mixed-criticality systems because it captures not only the real time requirement (meeting task deadlines) but also the requirement of mixed-criticality systems (meeting a high-criticality task deadline is more important than meeting deadlines of all lower-criticality tasks combined).

In this paper, we assume permanent node failures. Upon permanent failures of a set N_F of processors, we are interested in determining how to reallocate tasks, especially failed tasks, to healthy processors. We aim at using a small number of task reallocations to retain a good ν value for the mixed-criticality system. To support fault recovery, a task reallocation algorithm is invoked when any permanent failures of processors are detected. The reallocation algorithm takes the current task partitioning and the set of failed processors as inputs and calculates the reallocations and the resultant ν value.

IV. Task Reallocation Algorithms

In this section, we present our algorithms for task reallocation after node failures in mixed-criticality systems. Due to the high overhead of reallocating mixed-criticality tasks after node failures, our efficient task reallocation algorithm considers both the performance of mixed-criticality system and the number of reallocations. In case of node failures, our algorithm ensures the performance of mixed-criticality systems by accommodating high-criticality tasks first. High-criticality tasks are quickly recovered to run on healthy processors and they are favored over low-criticality tasks. That is, if a processor is fully utilized, low-criticality tasks running on the processor may be reallocated to another processor or even simply removed to save space for high-criticality tasks. On the other hand, our algorithm reduces the number of task reallocations by not deciding the task assignment from scratch and respecting the current task allocation as much as possible. In a word, our algorithm makes a good trade-off between the number of task reallocations and the performance of the mixed-criticality system.

We also present and analyze two baseline algorithms which enable us to spotlight the key performance trade-offs. Baseline algorithm 1 tries to maintain the performance of the mixed-criticality system as much as possible. It achieves this by considering all tasks and all healthy processors in the system and applying *Compress-on-Overload Packing* (COP) algorithm [6] to decide a brand-new task assignment. This approach, however, may require a large number of task reallocations. Baseline algorithm 2 tries to minimize the number of task reallocations as much as possible. It achieves this by only reallocating failed tasks. But, in this case, a high-criticality task in a failed node may be discarded if no healthy processor is found to have enough spare space to accommodate the task. In general, if there is no restriction on the number of task reallocations, we have the maximum flexibility and can adapt the task assignment as much as needed to achieve the optimal performance. However, the system performance does not increase linearly with the number of task reallocations. As will be shown, if correct tasks are chosen to be reallocated, we can achieve very good system performance by only a small number of task reallocations.

A. Baseline Algorithm 1

Baseline algorithm 1 is similar to the original COP algorithm presented in [6] which aims at minimizing the deadline misses of high-criticality tasks. It applies the two-phase task allocation [6] to assign tasks to healthy processors. In the first phase, the task set is sorted in decreasing order of criticality and *overload* utilization ($\frac{C_i^o}{T_i}$) (Line 1). For assigning a task, the algorithm considers healthy processors in $N \backslash N_F$ set in decreasing order of processor's overloaded

Algorithm 1: Task Reallocation Baseline 1

Input: Set of tasks Γ, Set of processors N, Current Task Partitioning M, Set of failed processors N_F

Output: New Task Partitioning M', Task Reallocations δ, and ν value

1 *Sort task set* Γ *in decreasing order of criticality and overload utilization*
2 **foreach** *processor* $PE_i \in N \backslash N_F$ **do**
3 **foreach** *task* $\tau_j \in \Gamma$ **do**
4 $Allocate(\tau_j, PE_i)$
5 $U_i = GetOverloadUtil(PE_i)$
6 **if** $U_i > threshold$ **then**
7 $Remove(\tau_j, PE_i)$
8 **end**
9 **else**
10 $Remove(\tau_j, \Gamma)$
11 $M'(\tau_j) = PE_i$
12 **end**
13 **end**
14 **end**
15 *Sort task set* Γ *in decreasing order of criticality and normal utilization*
16 **foreach** *processor* $PE_i \in N \backslash N_F$ **do**
17 **foreach** *task* $\tau_j \in \Gamma$ **do**
18 $Allocate(\tau_j, PE_i)$
19 $U_i = GetNormalUtil(PE_i)$
20 **if** $U_i > threshold$ **then**
21 $Remove(\tau_j, PE_i)$
22 **end**
23 **else**
24 $Remove(\tau_j, \Gamma)$
25 $M'(\tau_j) = PE_i$
26 **end**
27 **end**
28 **end**
29 $\delta = CalculateReallocations(M, M')$
30 $D = CalculateDuctilityMatrix(M')$
31 *Calculate* ν *value based on derived* D

fullness level $\sum_{\forall j} \frac{C_j^o}{T_j}$ (Lines 2-14). In the second phase, the task set is sorted in decreasing order of criticality and *normal* utilization ($\frac{C_i}{T_i}$) (Line 15). For assigning a task, the algorithm considers healthy processors in $N \backslash N_F$ set in decreasing order of processor's normal fullness level $\sum_{\forall j} \frac{C_j}{T_j}$ (Lines 16-28). After the two-phase allocation, if a task's new location turns out to be different from its old location, the task is reallocated and we then evaluate the system performance by calculating the number of task reallocations and the resultant ductility matrix and the corresponding ν value (Lines 29-31). Note, any remaining tasks unassigned by the algorithm are considered as tasks that always miss their deadlines in the ductility matrix.

B. Baseline Algorithm 2

The pseudo code of baseline algorithm 2 is given in Algorithm 2. Similar to Algorithm 1, it applies a two-

Algorithm 2: Task Reallocation Baseline 2

Input: Set of tasks Γ, Set of processors N, Current Task Partitioning M, Set of failed processors N_F

Output: New Task Partitioning M', Task Reallocations δ, and ν value

1 $\Gamma_F = \{\tau_j | M(\tau_j) \in N_F\}$
2 **foreach** *task* $\tau_j \in \Gamma \backslash \Gamma_F$ **do**
3 $M'(\tau_j) = M(\tau_j)$
4 **end**
5 *Sort failed tasks* Γ_F *in decreasing order of criticality and overload utilization*
6 **foreach** *processor* $PE_i \in N \backslash N_F$ **do**
7 **foreach** *task* $\tau_j \in \Gamma_F$ **do**
8 $Allocate(\tau_j, PE_i)$
9 $U_i = GetOverloadUtil(PE_i)$
10 **if** $U_i > threshold$ **then**
11 $Remove(\tau_j, PE_i)$
12 **end**
13 **else**
14 $Remove(\tau_j, \Gamma_F)$
15 $M'(\tau_j) = PE_i$
16 **end**
17 **end**
18 **end**
19 *Sort failed tasks* Γ_F *in decreasing order of criticality and normal utilization*
20 **foreach** *processor* $PE_i \in N \backslash N_F$ **do**
21 **foreach** *task* $\tau_j \in \Gamma_F$ **do**
22 $Allocate(\tau_j, PE_i)$
23 $U_i = GetNormalUtil(PE_i)$
24 **if** $U_i > threshold$ **then**
25 $Remove(\tau_j, PE_i)$
26 **end**
27 **else**
28 $Remove(\tau_j, \Gamma_F)$
29 $M'(\tau_j) = PE_i$
30 **end**
31 **end**
32 **end**
33 $\delta = CalculateReallocations(M, M')$
34 $D = CalculateDuctilityMatrix(M')$
35 *Calculate* ν *value based on derived* D

phase allocation. However, instead of considering and re-mapping all tasks like Algorithm 1, baseline algorithm 2 only takes failed tasks into account when reassigning task locations (Lines 1-4). In the first phase, failed tasks ($\{\tau_j | M(\tau_j) \in N_F\}$ where function M gives the processor initially assigned to run a task) are sorted in decreasing order of criticality and *overload* utilization ($\frac{C_i^o}{T_i}$) (Line 5). For each failed task, the algorithm tries to find a healthy processor with enough spare space to accommodate the task. Nodes are considered following the decreasing order of the overload fullness level. The algorithm calculates the resultant overload fullness level after adding the task to the candidate processor

and checks if it exceeds the schedulable threshold or not. If not, the task is reallocated to the processor. Otherwise, another candidate processor will be considered for the task. This procedure continues until either a processor is found for the task or all healthy processors have been tested. In the latter case, the task will be postponed until the second phase for consideration (Lines 6-18). In the second phase, unassigned failed tasks are sorted in decreasing order of criticality and *normal* utilization ($\frac{C_i}{T_i}$) (Line 19). This time, the algorithm checks candidate processors using their normal fullness levels (Lines 20-32). In the end, after the two-phase allocation, if a task's new location turns out to be different from its old location, the task is reallocated and we then evaluate the system performance by calculating the number of task reallocations and the resultant ductility matrix and the corresponding ν value (Lines 33-35). Note, any remaining tasks unassigned by the algorithm are considered as tasks that always miss their deadlines in the ductility matrix.

C. Efficient High-Ductility Task Reallocation Algorithm

Algorithm 3 presents the pseudo code of our new algorithm, which focuses on making a good trade-off between the number of reallocations and the performance of mixed-criticality system. It is based on the observation that high-criticality tasks should be allocated prior to low-criticality tasks in face of node failures. To follow this rule, failed tasks are simply sorted in the decreasing order of criticality so that high-criticality tasks are considered first (Lines 1-5). To favor a failed task with high criticality, lower-criticality tasks in the candidate processor are "preempted". That is, lower-criticality tasks are assumed to be removed when checking if the processor has enough space for the failed task. If the normal fullness level exceeds the schedulable threshold, then this processor cannot accommodate the failed task. Otherwise, the failed task will be allocated to the processor with all lower-criticality tasks removed. Following the decreasing order of criticality, these removed tasks may be added back to the processor if spare space exists. For those removed tasks that fail to be restored to the processor, they are put into the failed task list for reallocation. This procedure stops when either there is no tasks unassigned or there is no processor available to accommodate any task in the failed task list (Lines 6-39). In the end, after the task reallocation, we then evaluate the system performance by calculating the number of task reallocations and the resultant ductility matrix and the corresponding ν value (Lines 40-42). Note, any remaining tasks unassigned by the algorithm are considered as tasks that always miss their deadlines in the ductility matrix.

Algorithm 3: Efficient High-Ductility Task Reallocation

Input: Set of tasks Γ, Set of processors N, Current Task Partitioning M, Set of failed processors N_F
Output: New Task Partitioning M', Task Reallocations δ, and ν value

1 $\Gamma_F = \{\tau_j | M(\tau_j) \in N_F\}$
2 **foreach** *task* $\tau_j \in \Gamma \backslash \Gamma_F$ **do**
3 $\quad M'(\tau_j) = M(\tau_j)$
4 **end**
5 *Sort failed tasks* Γ_F *in decreasing order of criticality*
6 **foreach** *task* $\tau_j \in \Gamma_F$ **do**
7 $\quad$ **while** *unmarked processors exist in* $N/\ N_F$ **do**
8 $\quad\quad PE_i =$ *an unmarked processor from* $N/\ N_F$
9 $\quad\quad$ *mark* PE_i
10 $\quad\quad \Gamma_{LC_i} = \{\tau_k | M(\tau_k) = PE_i \ \&\& \ \varepsilon_k > \varepsilon_j\}$
11 $\quad\quad$ *remove* $M'(\tau_k)$ *value for each task* τ_k *in* Γ_{LC_i}
12 $\quad\quad Allocate(\tau_j, PE_i)$
13 $\quad\quad U_i = GetNormalUtil(PE_i)$
14 $\quad\quad$ **if** $U_i > threshold$ **then**
15 $\quad\quad\quad Remove(\tau_j, PE_i)$
16 $\quad\quad\quad Restore(\Gamma_{LC_i}, PE_i)$
17 $\quad\quad\quad$ *restore* $M'(\tau_k)$ *value for each task* τ_k *in* Γ_{LC_i}
18 $\quad\quad$ **end**
19 $\quad\quad$ **else**
20 $\quad\quad\quad Remove(\tau_j, \Gamma_F)$
21 $\quad\quad\quad M'(\tau_j) = PE_i$
22 $\quad\quad\quad$ **foreach** *task* $\tau_k \in \Gamma_{LC_i}$ **do**
23 $\quad\quad\quad\quad Allocate(\tau_k, PE_i)$
24 $\quad\quad\quad\quad U_i = GetNormalUtil(PE_i)$
25 $\quad\quad\quad\quad$ **if** $U_i > threshold$ **then**
26 $\quad\quad\quad\quad\quad Remove(\tau_k, PE_i)$
27 $\quad\quad\quad\quad$ **end**
28 $\quad\quad\quad\quad$ **else**
29 $\quad\quad\quad\quad\quad Remove(\tau_k, \Gamma_{LC_i})$
30 $\quad\quad\quad\quad\quad M'(\tau_k) = PE_i$
31 $\quad\quad\quad\quad$ **end**
32 $\quad\quad\quad$ **end**
33 $\quad\quad\quad \Gamma_F = \Gamma_F \cup \Gamma_{LC_i}$
34 $\quad\quad\quad$ *Sort failed tasks* Γ_F *in decreasing order of criticality*
35 $\quad\quad\quad break$
36 $\quad\quad$ **end**
37 $\quad\quad$ *unmark all processors in* $N/\ N_F$
38 $\quad$ **end**
39 **end**
40 $\delta = CalculateReallocations(M, M')$
41 $D = CalculateDuctilityMatrix(M')$
42 *Calculate* ν *value based on derived* D

V. Evaluation

Evaluation of the task reallocation algorithms are performed using a simulation tool we have developed to implement Compress-on-Overload Packing (COP) algorithm and Zero-Slack Rate Monotonic (ZSRM) scheduling [8, 6] in multiprocessor mixed-criticality systems. A series of randomly generated task sets of different sizes are used. The size of the task set varies from 30 to 120. The number of processors considered in the experiment varies from one fifth of the number of tasks to one third of the number

of tasks. Task period is randomly generated to be either 100, 200, or 400 time units. Task deadline is assumed to be equal to its period. The task priority is set by following Rate Monotonic Scheduling (RMS) which means the shorter the period is, the higher the priority. The schedulability utilization bound for RMS is $n(2^{\frac{1}{n}} - 1)$ where n is the number of tasks assigned to a processor. Two criticality levels are considered in the simulations: high and low. It is assumed that there are more low-criticality tasks than high-criticality tasks. Specifically, the number of low-criticality tasks is three times of the number of high-criticality tasks. The normal worst case execution time is between 0.1 and 0.2 times of the task period, and the overload worst case execution time is between 0.2 and 0.3 times of the task period. The actual execution time is assigned to be equal to the worst case execution time in both overload and normal operating states.

Experiments are conducted in two steps. First, tasks are allocated to a given set of processors following COP algorithm [6]. After finishing task allocation, each processor schedules all tasks on it using ZSRM scheduling where different workload situations are assumed, and then ductility matrix D and ν are calculated. Second, task reallocation baseline 1, 2 and our new algorithm are implemented and evaluated. For a given set of processors, we always assume a permanent processor failure and all tasks hosting in the processor need to be reallocated by using either of the three algorithms. Two performance metrics are used for the evaluation, i.e., the number of task reallocations and the ν value. In order to evaluate the effectiveness of different task reallocation algorithms, a series of different numbers of processors are given to host a task set. After reallocating the tasks in the failed processor, each processor starts running its task set again and outputs its ductility matrix. A final ductility matrix is generated based on the outputs of all processors and is used to derive the ν value. The number of task reallocations and the ν value before and after a node failure are calculated and compared.

The performance comparison results of the three algorithms with respect to the number of task reallocations are presented in Figures 1(a)-1(c). They correspond to task sets with 30, 60, and 120 tasks respectively. As can be seen in the figures, nearly all tasks need to be reallocated using baseline algorithm 1. Specifically, there are more than 24 tasks out of 30 tasks reallocated in all cases and 27 out of 30 tasks in most cases. There are more than 55 tasks out of 60 tasks reallocated and more than 114 tasks out of 120 tasks reallocated in all cases. That is nearly more than 90 percent of tasks being reallocated. It is obvious that baseline algorithm 1 does not perform well in terms of the number of task reallocations. Baseline algorithm 2 can significantly decrease the number of task reallocations. It is almost fixed in all scenarios. Specifically, there are always around 5 task reallocations for all three task sets. This is because we always assume that processor 1 fails in our experiment and processor 1 is usually allocated 5 tasks before failure [1]. Since baseline algorithm 2 only needs to consider reallocating failed tasks, there is often about 5 task reallocations. Therefore, it can achieve a very good performance in terms of the number of task reallocations. Compared to baseline algorithm 1, our efficient high-ductility algorithm can also significantly reduce the number of task reallocations. It requires only slightly more task reallocations than baseline algorithm 2.

Figure 2 shows the performance comparison results of the three algorithms with respect to the ν value. From Figures 2(a) to 2(c), each figure shows the performance comparison of the ν value before and after a node failure using the two baseline algorithms and our new algorithm. From Figure 2(a), we can observe that our algorithm always has a better performance than baseline 2 in all 5 cases. From Figure 2(b), we can observe that our algorithm achieves a better or at least the same ν value as baseline 2 in all 9 cases. Figure 2(c) shows that our algorithm preforms better or at least the same as baseline 2 in all cases. Therefore, in general, our algorithm is better than baseline 2 in terms of the achieved ductility of mixed-criticality systems.

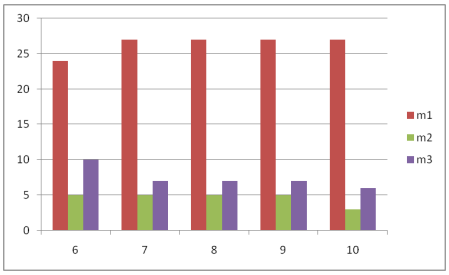
Compared to baseline 1 with respect to the ν value, our algorithm also shows a very good performance. From Figure 2(a), it can be observed that our algorithm performs better in 2 out of 5 cases while baseline 1 performs better in another 2 out of 5 cases. In the last case when 10 processors are used to host 30 tasks, both algorithms achieve the same ν value. From Figure 2(b), we can observe that our algorithm has the same performance as baseline 1 in 7 out of 9 cases. For the other two cases, our algorithm performs better in one case while baseline 1 performs better in the other. Figure 2(c) shows that both algorithms have the same performance in 13 out of 17 cases. For the other four cases, they again achieve a tie in performance with each algorithm performs better in two cases. Therefore, in general, our new algorithm achieves the same level of ductility as baseline algorithm 1 in mixed-criticality systems.

In summary, compared to baseline 2, our algorithm shows better performance in terms of the ν value with slightly more numbers of task reallocations. Compared to baseline 1, our algorithm shows the same performance in terms of the ν value with a far less number of task reallocations. Therefore, based on the above detailed analysis of the experimental results, we can safely conclude that our efficient high-ductility algorithm is effective in reducing the number of task reallocations while keeping very good performance for mixed-criticality systems.

VI. Related Work

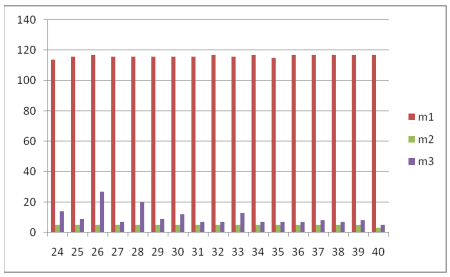
Mixed-criticality systems have been intensively studied recently driven by the need to consolidate functionalities of different criticalities onto a shared resource platform to reduce overall cost for development and certification [15, 7, 6]. The key issue in the design of mixed-criticality

[1]Since processor 1 is considered first when we allocate tasks with COP algorithm, it is often assigned the most number of tasks. Thus, in order to simulate the worst case scenario, we always assume that processor 1 fails. The algorithm itself, however, can be applied to resolve any processor crash failures.



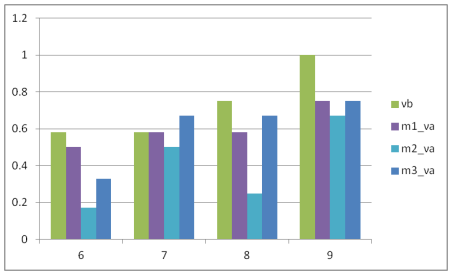
(a) Task Reallocations for 30 Tasks

(b) Task Reallocations for 60 Tasks

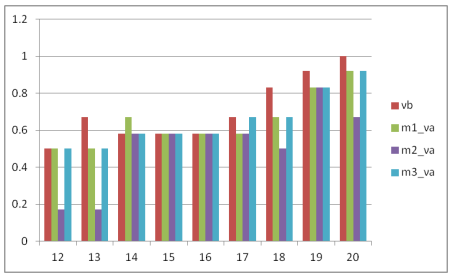


(c) Task Reallocations for 120 Tasks

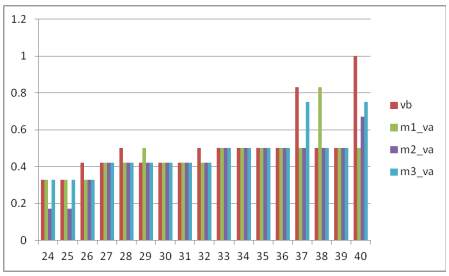
Fig. 1. The result of task reallocations using the two baseline algorithms and our algorithm. X axis represents the number of processors. Y axis represents the number of task reallocations. $m1$ is the result of task reallocations when using baseline algorithm 1. $m2$ is the result of tasks reallocations when using baseline algorithm 2. $m3$ is the result of task reallocations when using our algorithm.



(a) ν Value Comparison for 30 Tasks



(b) ν Value Comparison for 60 Tasks



(c) ν Value Comparison for 120 Tasks

Fig. 2. The ν value before and after a node failure by using the two baseline algorithms and our algorithm. X axis represents the number of processors. Y axis represents the ν value for multiprocessor mixed-criticality systems. νb is the ν value before the node failure. $m1\_\nu a$ is the result of ν value after the node failure when using baseline algorithm 1. $m2\_\nu a$ is the result of ν value after the node failure when using baseline algorithm 2. $m3\_\nu a$ is the result of ν value after the node failure when using our algorithm.

systems is the need to provide partitioning among applications in order to prevent interferences due to shared resource.

In order to ensure partitioning in mixed-criticality systems, different scheduling algorithms have been proposed. In [15, 7, 3], the authors focus on scheduling problems for certification. They argue that Certification Authorities (CA's) may have different correctness criteria with respect to a task. This difference in correctness criteria is expressed by different WCET estimates for a task in their proposed task model. In our case, we do not deal with scheduling for certification and instead we follow an approach similar to those in [8, 6]. Papers [8, 6] present an approach to guarantee that all high-criticality tasks meet their deadlines by stealing execution time from low-criticality tasks in case of insufficient resources. This approach is called asymmetric protection which prevents interference from low-criticality tasks to high-criticality tasks. However, besides asymmetric protection, we further investigate fault recovery and use task reallocation to increase the reliability of multiprocessor mixed-criticality systems.

Several papers have targeted at designing a fault-tolerant mixed-criticality system. Papers [2, 14] present a generic fault tolerant scheduling framework that allows every critical task to be feasibly replicated in both time and space dimensions. The basic idea is to derive feasibility window to reserve necessary slacks for re-execution of critical, highly-critical and ultra-critical task instances while a best-effort service is provided to non-critical task instances. Similarly, papers [5, 12, 1] use replication to provide fault tolerance for safety-critical tasks. Error propagation probabilities or failure probabilities are derived to measure the fault tolerance requirement. Although these approaches are capable of tolerating a wide range of faults, they come either with a high cost due to a hardware replication (spatial redundancy) or with a poor utilization of resources due to re-execution (time redundancy). In this paper, we instead use task reallocation to provide fault recovery in case of permanent processor failures. The developed mechanism falls into the category of adaptive resource management in distributed real-time

systems [13].

One closely related work is providing fault tolerance in mixed-criticality systems by task migration [10, 11]. An online greedy approach was proposed to migrate safety-critical tasks to other healthy processors in response to permanent faults. Check-pointing with rollback recovery is used to handle transient faults. Mixed-criticality property are defined by whether a task has to tolerate permanent faults or transient faults. This definition, however, limits the distinguished characteristic of mixed-criticality systems: different criticalities are assigned to tasks according to their effects on the systems. Also, they do not take criticality inversion problem into account. Therefore, a high-criticality task may miss its deadline due to the interference from a low-criticality task. In our approach, high-criticality tasks are always favored over low-criticality tasks in case of permanent faults and asymmetric protection is always ensured. Emberson et al. [4] proposed a method to minimize the number of task migrations for real-time systems to ensure the quality of service when changing system modes. They, however, did not consider the mixed-criticality property and processor failures.

VII. Conclusion and Future Work

In this paper, we presented algorithms to address the issue of task reallocation in mixed-criticality systems for recovering permanent node failures. Specifically, we studied two baseline algorithms and developed a new algorithm that combines the benefits of the two. To evaluate the performance of the task reallocation algorithms, two metrics are used: the number of task reallocations and the ν value which is an important metric reflecting the desired property of mixed-criticality systems [6]. We examined and compared the algorithms under various task set and system configurations and results show that our algorithm is effective in reducing the number of task reallocations while keeping high performance for mixed-criticality systems. In the future, instead of permanent node crash failures, we will investigate efficient adaptive recovery approaches for different types of faults.

VIII. Acknowledgements

The authors acknowledge support from General Motors Global Research & Development and NSFC award 61070002.

References

[1] P. Axer, M. Sebastian, and R. Ernst. Reliability analysis for mpsocs with mixed-critical, hard real-time constraints. In *Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis*, pages 149 –158, 2011.

[2] H. Aysan, R. Dobrin, and S. Punnekkat. A cascading redundancy approach for dependable real-time systems. In *Proceeding of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 467 –476, 2009.

[3] S.K. Baruah, A. Burns, and R.I. Davis. Response-time analysis for mixed criticality systems. In *Proceedings of the 32th IEEE Real-Time Systems Symposium*, pages 34–43, 2011.

[4] P. Emberson and I. Bate. Minimising task migration and priority changes in mode transitions. In *Proceeding of the 13th Real Time and Embedded Technology and Applications Symposium*, pages 158–167, 2007.

[5] Shariful Islam, Robert Lindstrom, and Neeraj Suri. Dependability driven integration of mixed criticality sw components. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 485–495, 2006.

[6] K. Lakshmanan, D. de Niz, R. Rajkumar, and G. Moreno. Resource allocation in distributed mixed-criticality cyber-physical systems. In *Proceeding of the IEEE 30th International Conference on Distributed Computing Systems*, pages 169 –178, 2010.

[7] Haohan Li and S. Baruah. An algorithm for scheduling certifiable mixed-criticality sporadic task systems. In *Proceedings of the 31th IEEE Real-Time Systems Symposium*, pages 183 –192, 2010.

[8] Dionisio de Niz, Karthik Lakshmanan, and Ragunathan Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 291–300, 2009.

[9] John Rushby. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Technical report, DTIC Document, 2000.

[10] Prabhat Kumar Saraswat, Paul Pop, and Jan Madsen. Task migration for fault-tolerance in mixed-criticality embedded systems. *SIGBED Rev.*, 6(3):6:1–6:5, October 2009.

[11] Prabhat Kumar Saraswat, Paul Pop, and Jan Madsen. Task mapping and bandwidth reservation for mixed hard/soft fault-tolerant embedded systems. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 89–98, 2010.

[12] M. Sebastian and R. Ernst. Reliability analysis of single bus communication with real-time requirements. In *Proceeding of the 15th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 3 –10, 2009.

[13] Nishanth Shankaran, Nilabja Roy, Douglas C. Schmidt, Xenofon D. Koutsoukos, Yingming Chen, and Chenyang Lu. Design and performance evaluation of an adaptive resource management framework for distributed real-time and embedded systems. *EURASIP J. Embedded Syst.*, 2008:9:1–9:20, January 2008.

[14] A. Thekilakkattil, R. Dobrin, S. Punnekkat, and H. Aysan. Optimizing the fault tolerance capabilities of distributed real-time systems. In *Proceeding of the 14th IEEE Conference on Emerging Technologies Factory Automation*, 2009.

[15] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, pages 239 –243, 2007.