Mixed Criticality Scheduling in Fault-Tolerant Distributed Real-time Systems

Abhilash Thekkilakattil, Radu Dobrin and Sasikumar Punnekkat
School of Innovation, Design and Engineering, Mälardalen University, Sweden
{abhilash.thekkilakattil, radu.dobrin, sasikumar.punnekkat}@mdh.se

***Abstract*—Modern safety critical real-time systems are composed of tasks of mixed criticalities and the problem of scheduling them in a fault tolerant manner, on a distributed platform, is challenging. Fault tolerance is typically achieved by using redundancy techniques, most commonly in the form of temporal redundancy which involves executing an alternate task before the original deadline of the failed task. Additionally, studies like Zonal Hazard Analysis (ZHA) and Fault Hazard Analysis (FHA) may impose extra constraints on the re-executions, e.g., spatial separation of alternates, to improve reliability.**

In this paper, we present a method for scheduling mixed criticality real-time tasks on a distributed platform in a fault tolerant manner while taking into account the recommendations given by the reliability studies like ZHA and FHA. First, we use mathematical optimization to allocate tasks on the processors, and then derive fault tolerant and fault aware feasibility windows for the critical and non-critical tasks respectively. Finally, we derive scheduler specific task attributes like priorities for the fixed priority scheduler. Our method provides hard real-time fault tolerance guarantees for critical tasks while maximizing resource utilization for non-critical tasks.

***Keywords*-Mixed Criticality Scheduling;Fault-tolerance; Real-time Systems**

I. Introduction

The increasing trend of integrating functionalities with different levels of criticality on the same platform has introduced new challenges in real-time scheduling, particularly in providing fault tolerance guarantees and in certifying such mixed criticality systems by a certification authority. A mixed criticality real-time system typically consists of a set of real-time tasks that vary in their 'importance' in ensuring the correctness of the system, e.g., the successful execution of some tasks may be more important than the others. In general, we can classify the tasks as either critical or non-critical, based on the consequences of deadline misses; a deadline miss on a critical task can cause catastrophic consequences, while a deadline miss on a non-critical task can cause only a minor degradation of the service provided by the system. Integrating mixed criticality tasks on the same platform can be beneficial in various ways, particularly in reducing cost and energy consumption [1].

This work was partially supported by the Swedish Research Council project CONTESSE (2010-4276) and Swedish Foundation for Strategic Research project SYNOPSIS

978-1-4799-5026-3/14/$31.00

Most of these mixed criticality systems are safety or mission critical, and operate in environments which may introduce errors in system, e.g., when using COTS components, potentially leading to failures. Additionally, the critical tasks may need to satisfy varying reliability requirements to facilitate graceful degradation of the system in the event of failures. Graceful degradation of the system under failures can constitute a strong safety argument while getting the system certified by a certification authority. In order to make sure that the system degrades gracefully under failures, the critical tasks may require complex fault tolerance mechanisms. In classical real-time systems, fault tolerance is typically achieved by using temporal redundancy, i.e., by re-executing the failed tasks or executing an alternate task. However temporal redundancy alone may not be sufficient to guarantee reliability, for example if there are permanent faults in the system. In this case, the tasks can execute on a different processor while meeting their original deadlines. A combination of temporal and spatial redundancy techniques can be used to guarantee graceful degradation of the system when failures occur.

In many systems, unforeseen interactions of unrelated tasks can cause system failures e.g., two memory intensive tasks on the same processor may cause memory overloads. Hardware reliability studies like Functional Hazard Analysis (FHA) and Zonal Hazard Analysis (ZHA) are done for safety critical systems to ensure that the proposed redundancies on the hardware components, e.g., wires and communication sub-systems, indeed exist. Functional hazard analysis is first carried out on such systems which identifies various function related events that are of interest to the system and proposes a design criteria based on these events. Zonal hazard analysis is then used to analyze the physical locations of the components and interconnections, the possible system to system interactions and severity of potential hazards based on these factors, i.e., it ensures that unrelated redundant subsystems are not affected by common causes. Functional hazard analysis and zonal hazard analysis, when applied to real-time systems, may have a direct impact on the allocation and scheduling of the real-time tasks, introducing extra constraints in addition to the normal replication constraints. Such constraints may recommend, for example, varying number of recovery attempts for different tasks, allocation of particular tasks on particular nodes, allocation of replicas on different nodes and physical separation between replicas of different tasks. Integrating the

recommendations of FHA and ZHA while scheduling real-time tasks can improve the overall reliability of the system, as well as contribute to safety.

Reducing development cost of the system is a major agenda for most system designers. In mixed criticality systems, from a certification point of view, the system needs to be build considering the worst case scenario of only the critical tasks. Consequently, the system designer can schedule the potential re-executions of the critical tasks and the non-critical tasks in an overlapping manner, with the caveat that non-critical tasks are shed upon critical task failures to guarantee the critical task re-executions.

In this paper, we aim to address the problem of allocation and scheduling of mixed criticality hard real-time tasks on a distributed system, while taking into consideration the recommendations of reliability studies like functional and zonal hazard analysis, when done on software systems [2]. We extend the framework first presented in [3] and subsequently extended to include the optimization formulation in a workshop in [4].

The main advantages of our approach are:

1) Efficient handling of task criticalities using feasibility windows.
2) Improved processor utilization and hence cost reduction through optimization.
3) Fault tolerance strategies covering multiple fault types.
4) Graceful degradation of the system under faults.
5) Supports the development of certifiable fault-tolerant mixed criticality systems.

The rest of the paper is organized as follows: in Sections II and III we present the related work and the system model respectively. The problem statement is given in Section IV and the methodology is described in Section V. Our approach is illustrated by an example in Section VI, followed by conclusions and ongoing work in Section VII.

II. Related Work

There exist numerous works in the field of fault tolerant scheduling of real-time systems. Kopetz [5] has detailed the requirements of a fault tolerant real-time distributed system. Ghosh et. al [6] proposed a fault tolerant scheduling algorithm for aperiodic tasks in multi processor systems. Bannister and Trivedi [7] proposed a simple heuristic algorithm that evenly distributes the computational load of the tasks over the nodes. More recently, Islam et al. [8] proposed a heuristic approach to perform allocation by considering dependability and real-time constraints as well as communication efficiency. In [9], the authors presented an exact schedulability tests for fault tolerant real-time task sets.

Baruah et al [10] presented a formal model of mixed criticality work load and demonstrated the intractability of mixed criticality scheduling under it. Guan et al [1] proposed an algorithm for scheduling mixed criticality work loads. Burns and Baruah [11] proposed two techniques for scheduling mixed criticality systems under the assumption of timing failures. While these works focused on transient faults, in this paper, we consider transient faults as well as recommendations of reliability studies like zonal and fault Hazard analysis in scheduling mixed criticality work load in a fault tolerant manner.

In [12], the authors underline the need for a comprehensive zonal analysis on critical systems. Fenelon et al. [2] proposes a design criterion based on such a kind of study when undertaken on software systems. They have discussed the use of safety analysis techniques in providing inputs to the design assessment phase to adopt a suitable design strategy.

III. System Model

A. Computational Model

We consider a distributed real-time system with identical multi-processors that communicate over a reliable communication media and are synchronized by relatively loose synchronization algorithms implemented in the software. We denote the set of n tasks by $\Gamma = \{\tau_1, \tau_2, .., \tau_n\}$, where each task represents a real-time thread of execution. Each task τ_i has a period T_i and a known worst case execution time C_i such that $\frac{C_i}{T_i} \leq 1 \; \forall i \in [1, n]$. The replication requirements on Γ are specified by $R = \{r_1, r_2, \ldots r_n\}$, where $r_i \in [0, \lfloor \frac{T_i}{C_i} \rfloor - 1]$. Additionally, $M = \{m_1, m_2, \ldots m_n\}$, where $m_i \in [0, r_i]$, specifies the distribution requirement of τ_i, i.e., the number of different nodes to be used for its replication. For instance, $m_1 = 3$ indicates that three replicas of the task τ_1 needs to be allocated in three different processors. We also use a binary variable β_i to denote the criticality of the task τ_i; $\beta_i = 1$ if τ_i is a critical task and $\beta_i = 0$ if τ_i is a non-critical task. The main task is called the primary and its recovery is called a replica or an alternate.

The execution time of a replica of τ_i is denoted by $\overline{C_i}$, where $\overline{C_i} \leq C_i$. Consequently, we denote the set of critical tasks, primaries and alternates by $\Gamma_c = \Gamma_c^{pri} \cup \Gamma_c^{alt}$, and the set of non-critical tasks by Γ_{nc}. Additionally, we use τ_j^k to denote the k^{th} replica of τ_j. When $k = 0$, τ_j^k represents the primary if τ_j is a critical task and the task itself in case it is a non-critical task. LCM represents the least common multiple of all the time periods of the tasks in Γ. The release time of τ_j^k is given by rel_j_k and its deadline is given by dl_j_k.

We assume that the tasks have deadlines equal to their periods. Each task has three main operational stages. First stage is the input stage where the input data is received from sensors or other tasks. Second stage is the computation stage and the third stage is the output stage where the output is delivered to the next task in the task chain or to the environment as a system output. We also assume that a task τ_i can execute only on one processor at a time instant. Execution of error detection or error handling mechanisms such as sanity checks and re-execution of failed computations are considered as a part of the computation stage.

B. Error Model

In this paper, we assume that a task represents a unit of failure. This failure may be due to a value or a timing error that compromises the correctness of the system. We also assume another class of errors, which we refer to as *zonal errors*, that

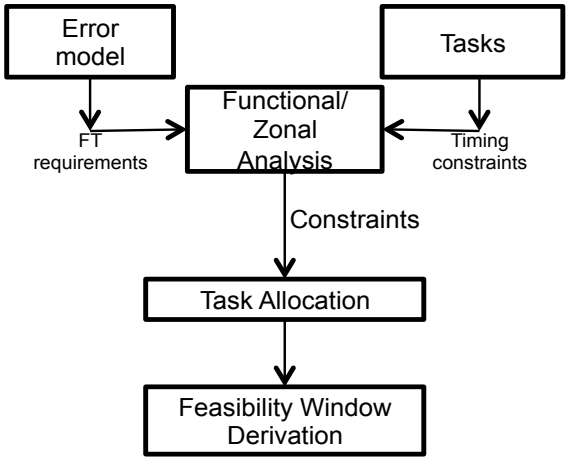

Fig. 1. Methodology overview

arises due to the unpredictable interactions of unrelated tasks, and is tolerated by ensuring physical separation of the recovery tasks. This is assumed to be identifiable at the design time by mechanisms such as Zonal hazard analysis [2]. We assume that each critical task has a known error frequency which can be determined with a high degree of confidence. We assume that the error detection is implemented in the underlying system and its worst case overhead is known. This error detection mechanism flags the errors as soon as it detects one. We assume that all the nodes have a consistent view of the errors flagged in the system. The recovery action is performed by the execution of a replica either in the same processor in case of a value/timing errors or in a different processor in case of a zonal error. It is assumed that during a recovery, the non-critical tasks are shed to execute the alternates.

IV. Problem statement

We assume a set of tasks Γ consisting of critical and non critical tasks $\Gamma_c \cup \Gamma_{nc} = \{\tau_1, \ldots, \tau_n\}$ with associated replication requirements $R = \{r_1, \ldots, r_n\}$, $r_i \in [0, \lfloor \frac{T_i}{C_i} \rfloor - 1]$ and distribution requirement $M = \{m_1, m_2, \ldots m_n\}$, $m_i \in [0, r_i]$. We want to find the contractual scheduling parameters for the tasks Γ such that:

1) the schedulability of the critical tasks and alternates, $\Gamma_c = \{\tau_i \mid r_i \neq 0\}$ is *guaranteed*
2) the schedulability of non critical tasks, $\Gamma_{nc} = \{\tau_i \mid r_i = 0\}$, is *maximized*
3) the utilization on processing nodes is maximized
4) the number of processing nodes is minimized

V. Methodology

Our goal is to derive the task parameters that reflect the criticality requirements of the real-time tasks that guarantees their successful execution before the deadline. While deriving the task parameters, there might be a need to split the real-time tasks into *artifact* tasks, in order to maximize the resource utilization per processor.

The task parameter derivation is performed in three steps:

A. Task allocation
B. Feasibility window derivation
C. Attribute derivation

We explain each of the steps in detail, in the following subsections:

A. Task Allocation

In this step, the real-time tasks are allocated to the multi-processors, considering the allocation constraints that result from studies like functional and zonal hazard analysis. The main constraints here are replication constraints, that specify the number of times the tasks have to re-execute before failing, and distribution constraints that require some of the re-executions to be carried out on a different processors. We use mathematical optimization to achieve an efficient allocation of the tasks to the processors, that satisfy the required constraints.

Let P_i be the variable that indicate whether the i^{th} processor is used. Our goal function is to minimize the number of processors:

$$G = \sum_{i=1}^{MAX} P_i, \qquad MAX = n + \sum_{j=1}^{n} m_j$$

The MAX represents the number of processors required to schedule the tasks and their last m_i replicas (that has to be scheduled on a different processor) exclusively to a processor each.

Let Pi_j_k be the binary variable that represents whether the k^{th} replica of the task τ_j (i.e., τ_j^k) is allocated to the i^{th} processor. Here, Pi_j_k can be either one or zero with one indicating that τ_j^k is allocated to processor P_i and zero otherwise.

$$\forall i \in [1, MAX],\ \forall j \in [1, n]\ and\ \forall k \in [0, r_j]$$

$$Pi\_j\_k \leq 1$$

Note that $r_j = 0$ for a non-critical tasks and the equation for the constraint remains valid.

If any one of $Pi\_j\_k = 1$, then the corresponding P_i is also 1, i.e., $\forall i \in [1, MAX]$,

$$P_i = \left\lceil \frac{\sum_{j=1}^{n} \sum_{k=0}^{r_j} Pi\_j\_k}{1 + \sum_{j=1}^{n} \sum_{k=0}^{r_j} Pi\_j\_k} \right\rceil$$

Here, we sum up the Pi_j_k's and divide the sum by one more than this sum and find the smallest integer greater than or equal to the result. In case the Pi_j_k's sum up to zero, the one in the denominator will make sure that a divide by zero error does not occur. If they do not sum up to zero, we get one since any number when divided by a larger number will result in a value between zero and one, the ceil of which will give a zero or one.

The release time of the primary of a task τ_j, given by rel_j_0, is set to the start of its period, i.e., $\forall j \in [1, n]$

$$rel\_j\_0 = 0$$

Similarly the release time of τ_j^k, given by rel_j_k, is set to the deadline of the previous replica i.e., τ_j^{k-1} for each critical task τ_j:

$$\forall j \in [1, n],\ \forall k \in [1, r_j]\ and\ \beta_j = 1$$

$$rel\_j\_k = dl\_j\_(k-1)$$

The minimum size of a window of the primary of a task or its replica, defined by a release time and deadline should be at least equal to the WCET of the task or its replica. We write this constraint as two constraints; one for the primary and the other for the replicas since the primary and the replicas can have different execution times and also because non-critical tasks do not have replicas. That is, $\forall j \in [1, n]$,

$$dl\_j\_0 \geq rel\_j\_0 + C_j$$

and,

$$\forall j \in [1, n],\ \forall k \in [1, r_j],\ and\ \beta_j = 1$$

$$dl\_j\_k \geq rel\_j\_k + \overline{C_j}$$

The deadline of the replica k of τ_j should not lead to an infeasibility in the derivation of a valid execution window for the replica $k+1$ of the same task.

$$\forall j \in [1, n],\ \forall k \in [0, r_j],\ and\ \beta_j = 1$$

$$dl\_j\_k \leq T_j - ((r_j - k) \times \overline{C_j})$$

Similarly, the deadline of a non-critical task must be less than or equal to the end of its period, i.e., $\forall j \in [1, n],\ and\ \beta_j = 0$:

$$dl\_j\_0 \leq T_j$$

The two constraints given above specifies an upper bound on the deadlines of a feasibility window. The first constraint bounds the deadline of a replica k such that there is provision to derive valid feasibility windows for replica $k+1$, of at least the minimum size which is equal to the execution time of the replica.

The next constraint ensure that the last m_j replicas of τ_j^k, are allocated to different processors.

$$\forall i \in [1, MAX],\ \forall j \in [1, n],\ and\ \beta_j = 1$$

$$Pi\_j\_0 + \sum_{k=r_j-m_j+1}^{r_j} Pi\_j\_k \leq 1$$

Here either Pi_j_0 or one of the last Pi_j_k's can be equal to 1.

The next constraint ensure that the primary and the replicas of a task are indeed allocated to one of the processors i.e., none of them remain unallocated.

$$\forall k \in\ [0, r_j],\ and\ \forall j \in\ [1, n]$$

$$\sum_{i=1}^{MAX} Pi\_j\_k\ =\ 1$$

The above constraint mandates that at least and only one of the Pi_j_k of τ_j^k is equal to one for $i \in [1, MAX]$.

The last set of constraints ensure that the processor utilization demand during any time interval does not exceed the length of the interval. This ensures the schedulability of the tasks allocated to each processor. The important problem here is to allocate the tasks in such a way that the non-critical tasks ($\beta_j = 0$) and the replicas of the critical tasks ($\beta_j = 1$) are allocated in a mutually exclusive manner. This is because non-critical tasks can be scheduled in overlapping windows with that of the replicas since the non-critical tasks are shed upon errors. This is achieved by two sets of constraints on the utilization of each processor: one ensuring that the total utilization of all critical tasks and its replicas scheduled on a processor is less than 100% for any time interval and the total utilization of critical and non-critical tasks scheduled on a processor is less than 100% in any time interval on the same processor.

The constraint while allocating the critical task primaries and their replicas is, $\forall i \in\ [1, MAX], t_1 < t_2$,

$$\sum_{\tau_x^z \in \Gamma_c} \eta_{x,z}(t_1, t_2) C'_x\ \leq\ t_2 - t_1,\quad \forall t_1, t_2$$

Where, $\eta_{x,z}(t_1, t_2)$ gives the number of instances of τ_x^z released in the interval $[t_1, t_2]$ and $C'_x = C_x$ when the primary of τ_x is considered and $C'_x = \overline{C_x}$ when its alternate is considered. The above constraint ensures that the processor utilization demand during any interval does not exceed the length of the interval while allocating the critical tasks' primaries and its replicas.

The next set of constraints ensure the schedulability of the critical tasks' primaries along with the non-critical tasks.

$$\forall i \in\ [1, MAX], t_1 < t_2$$

$$\sum_{\tau_x^0 \in \Gamma_c^{pri} \cup \Gamma_{nc}} \eta_{x,0}(t_1, t_2) C_x\ \leq\ t_2 - t_1,\quad \forall t_1, t_2$$

Where, $\eta_{x,0}(t_1, t_2)$ gives the number of instances of τ_x^0 released in the interval $[t_1, t_2]$.

B. Feasibility Window Derivation

In order to derive the attributes that guarantee schedulability, we first derive feasibility windows, which are temporal windows that provides offline or online guarantees to the tasks depending on their criticality.

We define two types of feasibility windows:

1) *Fault Tolerant* (FT) feasibility windows for critical tasks
2) *Fault Aware* (FA) feasibility windows for non-critical tasks

A Fault Tolerant Feasibility Window (FTW) is a temporal window in which a critical task has to complete its execution, such that it can feasibly re-execute (i.e., before its original deadline) upon an error. A Fault Aware Feasibility Window

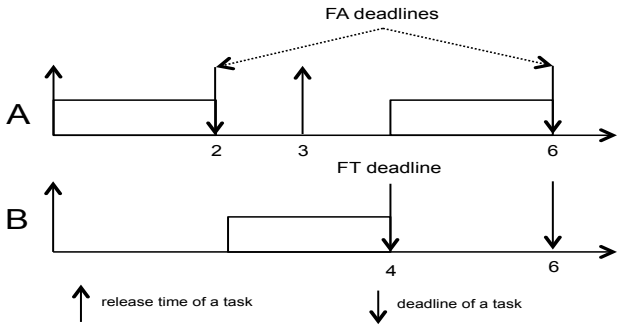

Fig. 2. FT and FA feasibility windows

(FAW) is a temporal window allocated to non-critical tasks, in order to control their interference with the critical ones, i.e., the execution of a non-critical task may not jeopardize the fault tolerant execution of any critical one. We use the method that we proposed earlier in [13] to derive the feasibility windows.

Let us consider 2 tasks A and B, where A is a non-critical task and B is a critical task. Let the time period and execution time of A be 3 and 2 respectively and that of B be 6 and 2. Let us assume that the maximum number of re-executions required by B is 1.

a) Fault Tolerant Feasibility Window Derivation: The latest time at which the alternate of task B should start executing to enable one feasible execution is given by its deadline minus the worst case execution time of the alternate. Since, according to our assumption, the WCET of the alternate is no greater than the WCET of the primary, the alternate of task B must start executing at time $t = 6-2 = 4$ to guarantee its successful execution. Hence the FT feasibility window of the primary of B is given by the interval, $(0, 4]$ and that of its alternate is given by the interval $(4, 6]$ as shown by the figure.

b) Fault Aware Feasibility Window Derivation: To derive the FA feasibility window for task A, we first schedule the primary of b to execute as late as possible and schedule A in the remaining slack. The figure shows the FA deadlines of task A. Hence the FA-feasibility window of task A is $(0, 2]$ for its first job and $(3, 6]$ for its second job.

In some cases, it might not be possible to derive FA-feasibility windows for the non-critical tasks. In these cases, the non-critical tasks are assigned with their original feasibility windows, given by their original release times and deadlines, and while deriving task priorities, they are assigned a background priority so that they do not influence the critical task executions.

C. Attribute Derivation

Once the feasibility windows are derived for every task, we need to derive attributes that guarantee the task executions before its deadline. Remember that critical tasks and their re-executions are provided offline guarantees, while the non-critical tasks are provided online guarantees. Since, we assume an FPS scheduler, we use integer linear programming presented [13] to derive the task priorities. During the priority derivation, there might be a need to split the tasks into artifact tasks, to maximize resource utilization. The use of integer linear programming ensures that the number of such splits is kept to the minimum. In cases where no valid feasibility windows were derived for the non-critical tasks, the tasks are assigned a background priority. These non-critical tasks execute only if there are enough resources at runtime e.g., in a better than worst case error scenario. Hence, the derived task parameters guarantee critical tasks and their re-executions offline, and provide the non-critical tasks with better levels of service whenever the runtime scenario permits.

VI. Example

We illustrate the proposed methodology by a simple example. Consider a set of tasks Γ={A,B,C,D} with parameters specified in Table I. We denote the k^{th} alternate of the task

TABLE I
SET OF TASKS- EXAMPLE

τ_i	T_i	C_i	R_i	M_i	$U_i(pri+alt)$	β_i
A	10	2	2	1	0.6	1
B	5	2	1	1	0.8	1
C	5	1	0	0	0.2	0
D	10	6	0	0	0.6	0

A and B by A_k and B_k. In our example, task A has T_i=10, C_i=2, R_i=2, M_i=1, which means that, the last of A's alternate needs to be executed in a different node than its primary.

The optimal allocation of the critical primaries, its alternates and the non-critical tasks in Fault Tolerant and Fault Aware Feasibility Windows, will result in a fault tolerant schedule in the minimum number of processors. The allocation of the FT/FA feasibility windows and the task executions in the worst case error occurrence scenario, is presented in figure 3.A. In this scenario, A_0, B_0 and $A_1$are hit by faults. Hence, the non-critical task D is shed due to the temporary overload on node 2, while C can still feasibly execute. At runtime, however, it is unlikely that the worst case scenario will occur. Consider that only the primary of A, i.e., A_0 is hit by an error in addition to the error on B_0. In this case, A_1 successfully execute as a result of which the execution of A_2 is no longer required. This creates the sufficient slack for the execution of task D, as illustrated in Figure 3.B, as a result of which D can execute successfully. Note that the fault tolerant and fault aware windows of the tasks in Figure 3.B are the same as in Figure 3.A.

In any case, the execution of the critical primaries and alternates are guaranteed feasible execution on the minimum number of nodes.

VII. Conclusions and ongoing work

In this paper we present an approach for scheduling mixed criticality real-time systems, by providing real-time guarantees for the critical tasks offline, while ensuring flexibility for the non-critical tasks. Mathematical optimization is used to

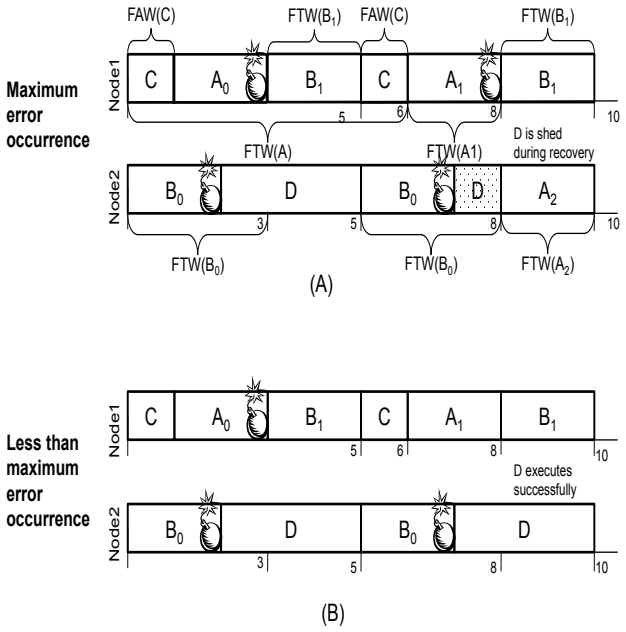

Fig. 3. (A). Allocation and execution under worst case error scenario (B). Allocation and execution under average case error scenario

derive optimal feasibility windows for task executions and re-executions while minimizing resource usage. The re-execution requirements are derived from studies like Functional Hazard Analysis and Zonal Hazard Analysis to maximize safety and reliability.

Future works will focus on the implementation of the proposed approach and comparison with other approaches, extensions to incorporate energy aware mechanisms and migration of non-critical tasks to gain even better levels of service.

REFERENCES

[1] N. Guan, P. Ekberg, M. Stigge, and W. Yi, "Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems," in *The IEEE International Real-Time Systems Symposium*, December 2011.

[2] P. Fenelon, J. A. McDermid, M. Nicolson, and D. J. Pumfrey, "Towards integrated safety analysis and design," *SIGAPP Appl. Comput. Rev.*, 1994.

[3] A. Thekkilakattil, R. Dobrin, S. Punnekkat, and H. Aysan, "Optimizing the fault tolerance capabilities of distributed real-time systems," in *14th International Conference on Emerging Technologies and Factory Automation, WiP*, 2009.

[4] A. Thekkilakattil, H. Aysan, and S. Punnekkat, "Towards a contract-based fault-tolerant scheduling framework for distributed real-time systems," in *The 1st International Workshop on Dependable and Secure Industrial and Embedded Systems*, June 2011.

[5] H. Kopetz, "On the fault hypothesis for a safety-critical real-time system," in *Automotive Software Connected Services in Mobile Networks, Lecture Notes in Computer Science*, 2006.

[6] S. Ghosh, R. Melhem, and D. Mosse, "Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems," *IEEE Transactions on Prarallel and Distributed Systems*, 1997.

[7] J. A. Bannister and K. S. Trivedi, "Task Allocation in Fault-Tolerant Distributed Systems," *Acta Informatica, Springer-Verlag*, 1983.

[8] S. Islam, R. Lindstrom, and N. Suri, "Dependability driven integration of mixed criticality sw components," *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, 2006.

[9] A. Burns, R. Davis, and S. Punnekkat, "Feasibility analysis of fault-tolerant real-time task sets," 1996.

[10] S. Baruah, H. Li, and L. Stougie, "Towards the design of certifiable mixed-criticality systems," in *16th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2010.

[11] A. Burns and S. Baruah, "Timing faults and mixed criticality systems," in *Dependable and Historic Computing*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011.

[12] R. Caldwell and D. Merdgen, "Zonal analysis: the final step in system safety assessment [of aircraft]," 1991.

[13] R. Dobrin, H. Aysan, and S. Punnekkat, "Maximizing the fault tolerance capability of fixed priority schedules," in *The 14th IEEE Internationl Conference on Embedded and Real-Time Computing Systems and Applications*, 2008.