Mixed Criticality Systems: Beyond Transient Faults

Abhilash Thekkilakattil[1], Alan Burns[2], Radu Dobrin[1], and Sasikumar Punnekkat[3]

[1]Mälardalen Real-Time Research Center, Mälardalen University, Sweden
[2]Real-Time Systems Research Group, Department of Computer Science, University of York, UK
[3]Department of Computer Science and Information Systems, Birla Institute of Technology and Science, India

***Abstract*—Adopting mixed-criticality architectures enable safe sharing of computational resources between tasks of different criticalities consequently leading to reduced Size, Weight and Power (SWaP) requirements. A majority of the research in mixed-criticality systems focuses on scheduling tasks whose Worst Case Execution Times (WCETs) are certified to varying levels of assurances. If any given task overruns its WCET, the system switches to a higher criticality and all the lower criticality tasks are discarded to make time for the execution of higher criticality tasks. Task execution time overruns are transient faults that are typically tolerated by simply executing an alternate task before the original deadline, or, by discarding the failed task to prevent it from interfering with higher criticality tasks. However, permanent faults such as processor failures can render the system to be useless, many times leading to unsafe states. In this paper we present a taxonomy of fault tolerance techniques to tolerate permanent faults, as well as map it to real-time mixed-criticality requirements based on the extend of fault coverage that in turn influences the associated assurance.**

I. Introduction

The complexity of software in real-time applications is increasing unlike ever before. This has created novel challenges in the design and verification of such systems, particularly with respect to timeliness guarantees. One way of reducing the complexity involved in providing temporal guarantees is to adopt a mixed criticality architecture, where only some tasks that are deemed to be critical to the operation of the system are provided strong guarantees, while others are typically provided weaker guarantees. The two main motivations behind adopting a mixed criticality architecture are i) efficient use of computational resources by integrating functionalities of varying criticalities (that are mostly certified to varying assurance levels) on the same platform [13] and ii) enable easy certification of systems by different certifying authorities [4]. Note that in this paper, ”resources” refer to ”computational resources” *e.g.,* a processor.

Mixed criticality systems has received good reception in the real-time systems community since Vestal described the problem in [13] (though some works such as [9] predates Vestal's work). The central assumption behind the mixed-criticality model proposed by Vestal [13] is that the individual task Worst Case Execution Times (WCET) monotonically increase with criticality of the components, because, the more critical a task is with respect to the correct functioning of the system, more conservative is its WCET estimate thereby increasing confidence on the estimates. We refer to all the mixed criticality scheduling models (*e.g.,* [13][4]) that adheres to this assumption as *Vestal's model*.

Most of the research in *mixed-criticality real-time scheduling* (which is Vestal's model) considers, what is referred to in the dependability community as the, *transient faults*. A *transient fault* can be a task execution time overrun (as assumed by Vestal's model) or single event upsets (as assumed by more recent works on mixed-criticality systems [10] [12]). Transient faults on real-time tasks are commonly tolerated by a simple re-execution of the same task or by an execution of an alternate task (or even by discarding the failed task so as not to jeopardize timeliness guarantees of other tasks, similar to Vestal's model). Although the ultimate goal of mixed-criticality systems is to provide higher *assurances* (*e.g.,* reliability) to higher criticality components, the focus of the majority of research is limited to transient faults, in particular to transient faults caused by different levels of Worst Case Execution Time (WCET) assurances. Moreover, the majority of the works assume that lower-criticality tasks can be safely discarded as the system switches to a higher criticality level (which is when tasks overrun their WCETs).

On the other hand, permanent faults can render a system to be completely useless *e.g.,* a processor failure. Moreover, inherent hardware faults (such as corrupted memory) can cause failures that are hard to detect by re-executions or executions of alternate tasks. In this case, system designers need to adopt spatial redundancy coupled with a voting mechanism to determine whether or not the generated outputs are erroneous. The majority of the research in mixed-criticality systems [7] do not consider the possibility of permanent faults, the issues/problems arising out of this, as well as its implications in the context of a mixed-criticality architecture.

Adopting spatial redundancy brings forth many challenges in conjunction with mixed-criticality architectures, particularly with respect to satisfying the ”S” and ”W” of the SWaP (Size, Weight and Power) constraints. One of the goals of adopting a mixed criticality architecture is to enable *safe* sharing of hardware resources between highly critical and lesser critical software components in order to reduce SWaP. However, spatial redundancy techniques require spare hardware that increases SWaP. Consequently, one of the goals of this paper is to investigate the possibilities of implementing a mixed-criticality architecture when using spatial redundancy.

Another essential requirement for guaranteeing ”timeliness” of real-time tasks that are replicated on a specified number of processors is that the hardware on which these replicas execute need to be tightly synchronized.

Traditionally, this was achieved by implementing tight synchronization schemes that typically require hardware modifications/support. Implementing tight synchronization schemes for all the spatially redundant components increases development costs, as well as make the components dependent on a global time base. The use of loosely synchronized systems (*i.e.,* systems where time synchronization is carried out by software) enables easier use of *e.g.,* multicore systems to implement spatial redundancy. Adopting a mixed-criticality architecture, especially on multicore systems, brings forth possibilities to implement the different functionality on software and perform voting using a ”time aware voter” (*e.g.,* see Aysan *et al.* [3]).

This paper presents a taxonomy of spatial redundancy techniques to tolerate permanent faults and identify how mixed-criticality architectures can be implemented when using spatial redundancy.

To summarize, the main contribution of this paper is a taxonomy of spatial redundancy techniques in the context of implementing mixed-criticality architectures. Particularly, we consider two main challenges:

- Maximizing efficiency of replica allocation to reduce cost. This can be achieved by provisioning resources based on the criticality of the associated tasks.
- Providing different levels of assurances for spatially redundant software components (tasks). The different levels of assurances are based on the extent of coverage of different faults.

The rest of the paper contains the system model in Section II, the main contributions of this paper in Section III followed by conclusions in Section IV.

II. System Model

In the following, we describe the system model in detail in order to make the context clearer.

A. Example System

In this paper, we use a running example of an autonomous vehicle adapted from Burns *et al.* [6]. The example services (tasks) are given in table I and consists of 2 high criticality functions, 4 medium criticality functions, 1 low criticality function and 1 non-critical function. These criticalities can be mapped to tasks associated with components that are assigned to specific Safety Integrity Levels (SILs) described in IEC 61508. Collision avoidance and braking control are high criticality tasks, while engine and lateral control, path finding and route planning can be considered to be of medium criticality. Display control functions are of low criticality while music streaming can be considered to be a non-critical task.

B. Fault Model

In this paper we consider both transient and permanent faults. As noted before, WCET overruns (such as the one assumed in the widely used Vestal’s model) are one type of transient faults and is tolerated by either aborting the task or by simply re-executing the task or executing an alternate task. Avizienis *et al.* [1] defines a permanent fault to be fault whose presence is assumed to be continuous in time. This can be for example, a processor failure in a distributed system that leads to an absence of output or a physical memory failure that leads to incorrect outputs. Permanent faults can be tolerated by employing spatial redundancy, *i.e.,* replicate the required functionality on multiple hardware.

Function	Criticality
Collision Avoidance	High
Braking Control	High
Engine Control	Medium
Lateral Control	Medium
Path Finding	Medium
Route Planning	Medium
Display Controls	Low
Music Streaming	Non-Critical

TABLE I: Autonomous Vehicle Example

Example II.1. *Consider the autonomous vehicle example given in table I that performs collision avoidance by detecting obstacles in its path and choosing appropriate actions such as either stop, slow-down or navigate around it. Object detection is a key functionality in this system. Suppose the processor on which the ”collision avoidance” software runs fails, it must be possible for the system to recognize the failure and bring the vehicle to a safe state or recover from it. A relatively easy way out is to replicate the collision avoidance functionality on multiple hardware,* e.g., *using Triple Modular Redundancy (TMR), and perform a voting. Adopting TMR enables tolerance to* 1 *fault,* i.e., *even if one of the replicas fail, in principle, the vehicle can detect it and still continue functioning.*

There are different types of permanent faults that need to be tolerated. The higher the coverage of the fault tolerance mechanism associated with a task, the higher the assurance one can give to the particular task. This provides for interesting trade-offs between task criticalities, development costs and SWaP.

Design faults: Occurs due to the deficiencies in the design and development of the system. A design fault may be due to the use of *e.g.,* a particular type of hardware or adoption of a specific implementation of a particular algorithm when building the system.

There are two types of design faults:

1) **Hardware Design faults:** Faults that either originate in the hardware or affects the hardware due to faulty design are referred to as hardware design faults. Examples of hardware design faults include manufacturing defects in the computer.
2) **Software Design faults:** Faults that affect the software of a computer system as a result of incorrect design are referred to as software design faults. Examples of software faults include faults due to incorrect interpretation of the specification, or a faulty implementation of an algorithm.

Random faults: A random fault is a fault whose time of occurrence cannot be predetermined, nor the causes can be identified offline. It may, for instance, be the result of wear and tear due to the continuous use of the system. On the other hand, the rate of occurrence of random faults for a given system can be estimated, for example, it is possible to analyze impact of wear and tear on the system.

Byzantine faults: Byzantine faults occur when some replicas behave arbitrarily differently. Moreover, different observers will record different behaviors of the replicas. In general, byzantine faults are the worst kind of faults and requires significant replication to be tolerated. Typically, to tolerate m byzantine faults, there is a need of $3m + 1$ replicas.

C. Fault Tolerance Mechanism

There are two primary types of fault tolerance mechanisms that can be implemented:

1) Fail Stop: In this form of fault tolerance, whenever a component fails, it stops functioning completely in order to prevent interfering from other (potentially dependent) components. Majority of the research in mixed-criticality real-time systems implement this form of fault tolerance in which, in case of an execution time overrun, the system switches to a higher criticality state and discards all the lower criticality components, essentially stopping all lower criticality tasks from executing.
2) Fail-Operational: In this form of fault tolerance, even if a component fails, the system continues to give an acceptable level of service by typically employing back-ups in the form of temporal or spatial redundancy.

D. Task Model

We consider a set of n real-time tasks/functions denoted by $\Gamma = \{\tau_1, \tau_2, \dots, \tau_n\}$ where each τ_i has a minimum inter-arrival time T_i and a deadline D_i. The transient faults on any task τ_i are tolerated either by simply re-executing τ_i or executing a back-up task to τ_i– we refer to both as *alternates*. In this context, the main execution of τ_i is referred to as the *primary* and has an execution time denoted by C_i. Permanent faults on any task τ_i are tolerated using spatial redundancy: each τ_i has a specified number of *replicas* that are executed in parallel, after which a voting is performed.

Every $\tau_i \in \Gamma$ needs to tolerate $\delta_i^t \geq 0$ transient faults and/or $\delta_i^p \geq 0$ permanent faults. This means that each τ_i requires δ_i^t *alternates* in addition to the primary execution and needs to be *replicated* on $(2\delta_i^p + 1)$ processors [11]. Alternately, the execution time of the back-ups of any task τ_i can be seen as the ”extra-time” that τ_i may need in case of a WCET overrun at any given criticality level.

Example II.2. *Consider a task τ_1 that needs to tolerate $\delta_i^t = 1$ transient fault and $\delta_i^p = 0$ permanent faults. This means that τ_1 needs to be re-executed once or an alternate task must be executed if there is a transient fault on τ_1.*

This model generalizes the widely used model for mixed-criticality systems, since *e.g.,* by enforcing $\delta_i^t \leq 1$ (and $\delta_i^t = 0$ *i.e.,* no replication) we get a dual criticality system: when $\delta_i^t = 1$, τ_i becomes a high criticality task that has two execution times (analogously one alternate) and when $\delta_i^t = 0$, τ_i becomes a low criticality task that can be safely discarded upon overrun. In the context of the widely used model, hereafter referred to as Vestal's model, the recovery task can be seen as the ”extra” duration for which high criticality tasks can execute in case of WCET overruns before the system switches to a higher criticality level.

III. Mixed-Criticality Design Challenges in Dependable Real-time Systems

In this section, we present a how different levels of assurances can be provided to tasks of different criticalities when building predictable mixed-criticality systems that employs spatial redundancy. We first explain the time synchronization problem when using spatial redundancy, that may lead to disasters in hard real-time systems. We then identify how mixed-criticality architectures can be implemented when using spatial redundancy to minimize SWaP requirements as well as the effort required to implement them.

Spatial redundancy, time synchronization and mixed-criticalities: Depending on the criticality of the task, the replicas of the task may be implemented as simple circuits developed by independent teams to ensure diversity. Alternately, the different replicas may be developed as software, by independent teams, and may be scheduled together with other tasks on processors from different vendors; albeit with reduced assurance when compared to the previous case. One of the key design challenge involved in providing spatial redundancy, in this case, is to ensure synchronization of different replicas to provide timely output to the voter. The need for tight synchronization is illustrated by the following simple example.

Example III.1. *Consider the sensors associated with lateral control of the example autonomous vehicle described in Table I. In order to provide high assurance to the Lateral Control function, spatial redundancy must be employed. Suppose the associated sensor is triplicated, with a voter, then there is a risk of two of the replicas giving the same output while the third gives a late output that is different because of the change in value over time. In this case, there is a risk that the voter discards the correct (albeit late) value by tagging it as ”incorrect” since it does not agree with the outputs from the two other sensors.*

These errors can be tolerated by adopting a tight time synchronization between the different replicas. However, providing tight synchronization is costly and requires significant effort. Moreover, having a tight synchronization between different replicas makes the system heavily dependent on the global time base. Alternately, for lesser critical tasks, loose synchronization algorithms implemented using software may be used that requires lesser development effort and cost. When using loose synchronization scheme, care must be taken to ensure that the voter does not suffer from timing errors. Loosely synchronous systems facilitate cost reduction by enabling the use of commercially available real-time operating systems on the individual processors without requiring modifications to enable tight synchronization. Each processor used to replicate the different tasks can execute the replicas using the local scheduling algorithm that can be, for example, EDF or FPS (as long as the individual replicas produce timely outputs). A time aware voter such as the one proposed by Aysan *et al.* [3] can detect and tolerate timing errors, consequently providing dependability guarantees.

A. Mixed-Criticality Architecture for Spatially Redundant Functions

Providing reliability and safety guarantees using spatial redundancy implies increased hardware that in turn results in increased SWaP requirements. Adopting a mixed-criticality architecture reduces SWaP requirements by provisioning the computing resources such that it reflects the task criticalities and the associated required assurance levels. Implementing mixed-criticality architectures for systems using spatial redundancy is still largely an unexplored area. In this section, we investigate methods to provide different levels of assurances to different tasks that uses spatial redundancy to improve reliability and safety while reducing SWaP. An overview of the mapping of the criticalities to tasks based on the fault coverage assurances is summarized in table II.

High Criticality Tasks: The high criticality tasks are the *most important* tasks in the system and require a very high level of assurance. Consequently, the probability of failures need to be significantly low. These tasks are associated with components classified as *e.g.,* SIL 4 of the IEC 61508 and are highly critical for the safe operation of the system. The high criticality task failures can result in disastrous consequences for the system and hence need to be provided with the highest level of assurance. For example, the collision avoidance and braking control in Table I are highly critical functionalities to ensure safe operation of an autonomous car.

Assurance Mechanism: The highest criticality tasks may be implemented on dedicated hardware to ensure isolation (as is typically done in many systems [14]), and a high integrity voter implemented as a simple electronic circuit performs voting. The use of simple electronic circuits implies that the voter can be verified to a very high degree of assurance [2], and the use of dedicated hardware guarantees that the tasks are protected from many types of faults.

- In order to provide assurances against random faults, the high criticality tasks are typically replicated *e.g.,* using Triple Modular Redundancy (TMR).
- To provide protection against design faults (both hardware and software), there is a need to ensure diversity. This can be done by N-version programming, *i.e.,* developing the different replicas using different development teams. Moreover, the different teams must use hardware and development tools from different vendors.
- To protect against byzantine faults, byzantine fault tolerance mechanisms must be adopted. Protection against byzantine faults imply further increase in hardware requirements. Typically, to tolerate δ_i^p byzantine faults, there is a need of $3\delta_i^p + 1$ replicas. Note that byzantine failures are observed more frequently than expected [8]. If the tasks are protected against byzantine faults, they are implicitly protected against all other faults (and hence does not require *e.g.,* TMR).
- The replication and diversity ensures that the system is tolerant to many transient faults since there is redundancy. Typically, by having $2\delta_i^p + 1$ redundant system implies protection against $2\delta_i^p$ transient faults. The Airbus A320, for example, uses both replication and diversity to ensure fault tolerance [5].

A key challenge here is to ensure tight synchronization between the replicas and the voter to ensure timeliness of the generated output. As a consequence of the above design, the development of high criticality tasks can be very costly.

Medium Criticality Tasks: The medium criticality tasks correspond to components of the system that are to be certified as *e.g.,* SIL 3 of IEC 61508. A failure in these tasks can cause serious consequences to the correct functioning of the system. However, consequences of medium criticality task failures are less disastrous than the high criticality failures, and the associated reliability guarantees need not be as high as the critical tasks (or the probability of a failure causing a disaster is lower compared to the high criticality tasks). Consequently, the medium criticality tasks need not be provided with the highest assurance level similar to the critical tasks. They can be provisioned less pessimistically, even using commercially available high integrity processors, than high criticality tasks to save on Size, Weight or Power. Engine control, given in Table I, is an example of a medium criticality task. Even though it may not be as critical as collision avoidance, there is a need to ensure its failure free execution.

Assurance Mechanism: The medium criticality tasks may be implemented in software on high integrity processors using *e.g.,* table driven scheduling. Even though the medium criticality tasks need not be implemented as electronic circuits to guarantee high assurance, they need to be made significantly fault tolerant.

- The medium criticality tasks can be replicated *e.g.,* using TMR to protect it against random faults. The replicas of the medium criticality tasks may be scheduled using highly predictable scheduling algorithm *e.g.,* table driven scheduling on different processors. These tasks need to execute in lock-step, and on completion pass on the output to a voter that then performs voting to mask any task failures.
- The medium criticality tasks can be made tolerant to design faults (both hardware and software) by ensuring diversity, *e.g.,* choosing processors on which the replicas execute from different vendors and by ensuring the use of N-version programming.
- The replication and diversity guarantees protection against many transient faults.

Such a setup requires tight synchronization schemes between the high integrity processors, in order to guarantee timeliness of the replica outputs and in turn guarantee timely output

Task Criticality	Transient Faults	Random Faults	Software Faults	Hardware Faults	Byzantine Faults
High	Fully covered	Fully covered	Fully covered	Fully covered	Fully covered
Medium	Fully covered	Fully covered	Fully covered	Fully covered	
Low	Fully covered	Fully covered	Fully covered	Partially covered	
Non-critical	Fully covered	Partially covered			

TABLE II: Mapping criticalities to tasks based on fault coverage.

from the voter. The advantage here is more than one medium criticality tasks can be scheduled on the same processor, as opposed to implementing them on the hardware, consequently reducing SWaP requirements. Since the tasks are implemented as software, and are scheduled on commercially available processors, the development cost associated with medium criticality tasks will be less.

Low Criticality Tasks: Low criticality tasks are associated with those components that need to be provided with *e.g.,* SIL 2 guarantees under IEC 61508 standard. Failures on low criticality tasks can cause less severe disruption of services in the system that are not disastrous (or the probability of a failure causing a disaster is low). However, these tasks are still required to ensure the normal operation of the system and needs to be provided with appropriate guarantees. Display controls in autonomous vehicles may not be as critical as collision avoidance or engine control and hence need not be provided with the same level of assurance.

Assurance Mechanism: The low criticality tasks may be implemented on commercially available multicore processors and scheduled using any standard real-time scheduling algorithm. However, some level of fault tolerance must be implemented to ensure the associated failure probabilities are low.

- The low criticality tasks may be replicated on different processors of the multicore platform. The different cores may be synchronized using relatively cheap synchronization algorithms, and a ”time aware” voter (*e.g.,* [3]) can guarantee timeliness of the generated output.
- Protection against software design faults can be implemented by employing N-version programming. Since, the replicas are scheduled on the different cores of the same multicore platform, no protection exists against hardware design faults and many hardware operational faults.
- The replication ensures that the system is automatically protected against many transient faults. Moreover, if software diversity is ensured, it further increases protection against many transient faults.

The use of commercially available processors, together with the possibility of adopting loose synchronization enables the use of relevant uniprocessor or multiprocessor scheduling algorithms such as Earliest Deadline First or Fixed Priority Scheduling to schedule the replicas, consequently enabling the use of commercial real-time operating systems. The main concern here is regarding the faults that may occur due to the loose synchronization. In this case, a time aware voter such as the one propsed by Aysan *et al.* [3] can guarantee absence of timing faults in such loosely synchronized systems. Adopting such an architecture enables efficient use of the available processing power since many algorithms, *e.g.,* EDF, that are known to be optimal can be employed.

Non-Critical Tasks: Non-critical tasks are the least ”important” tasks in the system as they are associated with components that can be given lowest level of assurance *e.g.,* SIL 1 guarantees of IEC 61508. Non-critical tasks can be safely discarded without affecting the normal operation of the system. The only major concern in this case is that the non-critical tasks must not ”interfere” with the execution of higher criticality tasks *i.e.,* they must be protected against transient faults by implementing a fail silent mechanism. Vehicular entertainment related task, such as music streaming in Table I, are good example of non-critical tasks that are not significant for the specified mission.

Assurance Mechanism: Non-critical tasks are scheduled normally along with tasks of higher criticalities. Non-criticality tasks may re-execute whenever there is slack, and is discarded immediately upon transient faults like an execution time overrun. The presence of extra computing resources ensures that non-critical tasks have a higher possibility of re-execution. A limited form of protection against transient faults and random faults can be added by ensuring that the non-critical tasks can re-execute whenever spare computing capacity is available.

Needless to say, depending on the coverage of faults, more criticalities can be defined.

IV. Conclusions

Even though the ultimate goal of mixed-criticality systems is to enable efficient resource usage while providing different levels of assurances to different components, the majority of research in mixed-criticality systems focus only on issues related to tolerating execution time overruns, which are only one type of transient faults. Despite the fact that more recent works considered transient faults other than execution time overruns, the challenges with respect to implementing mixed-criticality architectures for tolerating permanent faults has largely remained out of focus. The hardware redundancy required to tolerate permanent faults implies increased SwaP requirements, while mixed-criticality architectures enable hardware provisioning to real-time tasks based on the associated required assurance levels. In this paper, we present a taxonomy of spatial redundancy techniques, as well as propose a mapping of the assurance levels to task criticalities based on the extend of fault coverage with respect to permanent faults. A positive side effect of using spatial redundancy is that transient faults, such as execution time overruns, are automatically covered, and hence, this paper aims to initiate a discussion on the use of spatial redundancy techniques in the context of mixed-criticality systems.

Future work include investigation of optimal resource allocation strategies for assurances against different types of faults, as well as scheduling mechanisms for the tasks and their different replicas on multiprocessor platforms to guarantee timely outputs to the voter.

References

[1] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable Secure Computing*, January 2004.

[2] Huseyin Aysan. Fault-tolerance strategies and probabilistic guarantees for real-time systems. In *PhD thesis, Malardalen University*, June 2012.

[3] Hüseyin Aysan, Iain Bate, Patrick Graydon, and Sasikumar Punnekkat. Improving reliability of real-time systems through value and time voting. In *The 19th IEEE Pacific Rim International Symposium on Dependable Computing*, December 2013.

[4] S. Baruah, V. Bonifaci, G. D'Angelo, Haohan Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling real-time mixed-criticality jobs. *IEEE Transactions on Computers*, August 2012.

[5] D. Briere and P. Traverse. Airbus a320/a330/a340 electrical flight controls - a family of fault-tolerant systems. In *The Twenty-Third International Symposium on Fault-Tolerant Computing*, pages 616–623, 1993.

[6] A. Burns, D. Prasad, A. Bondavalli, F. Di Giandomenico, K. Ramamritham, J. Stankovic, and L. Strigini. The meaning and role of value in scheduling flexible real-time systems. *J. Syst. Archit.*, 2000.

[7] Alan Burns and Rob Davis. Mixed criticality systems - a review. In *Available:http://www-users.cs.york.ac.uk/ burns/review.pdf (accessed on 31 July 2015)*.

[8] Kevin Driscoll, Brendan Hall, Hkan Sivencrona, and Phil Zumsteg. Byzantine fault tolerance, from theory to reality. In *Computer Safety, Reliability, and Security*, Lecture Notes in Computer Science, pages 235–248. Springer Berlin Heidelberg, 2003.

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

[10] RisatMahmud Pathan. Fault-tolerant and real-time scheduling for mixed-criticality systems. *Real-Time Systems*, 2014.

[11] Sasikumar Punnekkat. *Schedulability Analysis for Fault Tolerant Real-time Systems*. PhD thesis, University of York, UK, June 1997.

[12] Abhilash Thekkilakattil, Radu Dobrin, and Sasikumar Punnekkat. Fault tolerant scheduling of mixed criticality real-time tasks under error bursts. In *The International Conference on Information and Communication Technologies*. Elsevier Procedia Computer Science, December 2014.

[13] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *The 28th IEEE International Real-Time Systems Symposium, 2007*, December 2007.

[14] K. Vipin, S. Shreejith, S.A. Fahmy, and A. Easwaran. Mapping time-critical safety-critical cyber physical systems to hybrid fpgas. In *The IEEE International Conference on Cyber-Physical Systems, Networks, and Applications*, 2014.