IMPROVING RELIABILITY AND LATENCY OF HIGH CRITICAL TASKS IN MIXED CRITICALITY SYSTEMS THROUGH TASK RESCHEDULING

by

Vamsi Krishna Karanam, BTech

THESIS

Presented to the Faculty of

The University of Houston-Clear Lake

In Partial Fulfillment

Of the Requirements

For the Degree

MASTER OF SCIENCE

in Computer Engineering

THE UNIVERSITY OF HOUSTON-CLEAR LAKE

December, 2019

IMPROVING RELIABILITY AND LATENCY OF HIGH CRITICAL TASKS IN MIXED CRITICALITY SYSTEMS THROUGH TASK RESCHEDULING

by

Vamsi Krishna Karanam

APPROVED BY

_____

Hakduran Koc, PhD, Chair

_____

Jian Lu, PhD, Committee Member

_____

Xiaokun Yang, PhD, Committee Member

RECEIVED/APPROVED BY THE COLLEGE OF SCIENCE AND ENGINEERING

_____

David Garrison, PhD, Associate Dean

_____

Miguel Gonzalez, PhD, Dean

Dedication

To

My Family

Acknowledgments

I would like to express my sincere appreciation and gratitude to all of those who have contributed to my master's thesis and those who have supported me throughout the entire process. I will always be grateful for that.

I would like to thank my supervisor Dr. Hakduran Koc for his excellent guidance and engagement throughout my thesis period. He has been supportive since the day I began working on my research. His insightful discussions and suggestions on the research helped me finish this thesis successfully. Moreover, I would like to thank my thesis committee, Dr. Jiang Lu, and Dr. Xiaokun Yang, for their encouragement and their helpful advice.

My greatest love and respect for my mother Dr. C. Madhumathi, father Lokanadha Naidu. I cannot thank them enough for their love and trust they had in me through all these years. I would not have made it this far without them. Loads of love and thanks to my sibling Sai Krishna, who stood by me in my times of need and helped work through rough patches.

I would like to thank my friends, Deepthi, Raveesh, Greeshma, Anupama, Lohit, for their constant support and for making life more fun and enjoyable. Thanks to my roommates Aslesh and Bhargav for bearing with me for the last two years. Thanks to my colleagues at Indian Students Association ('17, '18, '19) and my fellow research students for making my student life an interesting journey so far.

ABSTRACT

IMPROVING RELIABILITY AND LATENCY OF HIGH CRITICAL TASKS IN MIXED CRITICALITY SYSTEMS THROUGH TASK RESCHEDULING

Vamsi Krishna Karanam
University of Houston-Clear Lake, 2019

Thesis Chair: Hakduran Koc, PhD

A Mixed Criticality System (MCS) consists of various hardware and software components executing tasks with different criticality levels. The criticality of a task is determined by its impact on the overall system output (e.g., safety critical and mission critical tasks or low critical and high critical tasks). In today's world, such systems can be found almost everywhere including cars, airplanes, remotely piloted vehicles and so on. An MCS needs to be designed considering different criticality scenarios depending on the requirements of the operating environment. Significant amount of research has been dedicated to improve various parameters of MCSs such as reliability, performance, and power consumption.

In this thesis, we focus on improving the reliability and execution latency of high critical tasks in mixed criticality systems running on a Hardware/Software codesign environment. The system can run in two different operating modes: low criticality mode, which is the normal operating mode of the system, and high criticality mode. We propose two different algorithms: Reliability Priority Algorithm and Latency Priority Algorithm. In reliability priority approach, the algorithm schedules all tasks in the system and returns the final reliability and the latency of the system in low criticality operating mode. In high criticality mode, the algorithm gives priority to high critical (HC) tasks over low critical (LC) ones during the scheduling process. The LC tasks are scheduled in the gaps available considering the latency constraints. The algorithm returns the overall reliability of the system and the reliability of the HC tasks in both modes of operation. In latency priority approach, we prioritize the execution latency of HC tasks over their reliability. In the low criticality mode, the algorithm schedules the tasks to the fastest components available at the point of arrival. In the high criticality mode, HC tasks are scheduled before LC tasks in order to improve the latency of HC tasks.

The results of the experimental evaluation clearly show the viability of the proposed algorithms. The reliability priority algorithm increases the reliability of the HC tasks by 6.6% on the average and the latency priority algorithm improves the execution latency of the HC tasks by 23.8% on the average for the automatically generated task graphs.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I:

INTRODUCTION

In today's world, we are surrounded by electronically controlled systems enhancing our daily lives. Mixed Criticality Systems (MCS) are predominantly becoming a part of modern-day systems. These systems contain various hardware and software components executing tasks of different criticality such as safety critical and mission critical. Such systems can be found almost everywhere including cars, airplanes, remotely piloted vehicles (whether a toy or a tool). A significant amount of research has been dedicated to improving various parameters of MCSs such as reliability, performance, and power consumption [5, 6].

Task criticality is an important driver in real time systems in order to effectively utilize the limited processing elements with tight constraints. Classifying tasks as High Critical (HC) or Low Critical (LC) is also important for scheduling tasks efficiently to ensure available computing power. While LC tasks do not have a significant impact on the system performance, HC task execution is the deciding factor of the quality of the result and even affect the correctness of a result. This directs the research into treating the tasks with separate approaches based on their criticality such as dual criticality of a system. With dual criticality scheduling, the system schedules tasks based on their criticality levels within a given latency constraint. The criticality of a task is subject to change. It can change with the focus of the system and with the time the system has been in operation. Tasks changing the criticality levels in the course of execution bring additional challenges to task scheduling. The system must consider the change and adjust the operation to cater to the HC tasks. The system used this work is designed to operate in two execution modes. In low criticality mode, all tasks are treated to be of equal criticality, and they are scheduled accordingly in order to meet the design constraints. In

high criticality mode, the HC tasks are held in priority and some LC tasks may be excluded from scheduling in order to achieve the highest possible value for the optimization metric targeted.

The approach presented in this thesis partitions the tasks as high and low critical ones and prioritize HC tasks. This allows the system to schedule tasks based on either reliability or latency when needed while also moving between high criticality and low criticality operation modes as dictated by the system requirements. The execution latency and reliability of tasks are provided in the form of a technology library to the algorithms which start scheduling process by choosing the tasks to schedule based on the predefined rules of priority and assigning them to the appropriate processing elements. These rules take into consideration the number of tasks dependent on the current task, criticality, and the distance to the sink node to compare the priority of the task in question. In case of a tie between tasks in all conditions stated above, the task that can run with the highest performance in the available processing element is considered.

More specifically, in low criticality mode, the algorithm schedules all tasks in the system and returns with the final reliability and the latency of the system. In high criticality mode, the algorithm first considers the HC tasks during the scheduling process. Only after assuring that all HC tasks have been scheduled, the LC tasks are considered for scheduling. The LC tasks are scheduled in the gaps generated by the HC tasks and can be run till all of them are scheduled or a few LC tasks can be excluded if the overall latency exceeds the latency constraint. The algorithm returns the overall reliability of the system and the reliability of the HC tasks in both the modes of operation. By prioritizing HC tasks, we intend to assure the survival of the system by ensuring the completion of HC tasks as early as possible, and then, incorporating the LC tasks. This set of algorithms

prioritizes the reliability of the system over its execution latency, so we refer it as Reliability Priority approach.

A similar approach is utilized to formulate another set of algorithms that prioritize the execution latency of High Critical tasks over their reliability hereby referred to as Latency Priority approach. This approach intends to execute the tasks at a quicker pace in order to be able to implement other reliability measures such as duplication in the slack generated. The low criticality mode of the system accepts the data of each task excluding criticality and schedules the tasks to the fastest components available at the point of arrival. This results in a schedule faster than the reliability approach but at a cost of performance.

The high criticality mode in Latency Priority approach processes HC tasks ahead of LC tasks in order to improve the latency of HC tasks. LC tasks are scheduled in the time leftover or some are dropped if they cannot be accommodated within the constraints imposed on the system.

The experimental evaluation conducted using automatically generated task graphs clearly shows that the proposed algorithms successfully improves the reliability and the execution latency of the HC tasks in the system. More specifically, the Reliability Priority approach increases the reliability of the HC tasks and the Latency Priority approach improves the execution latency for the HC tasks in the system.

CHAPTER II:

REVIEW OF LITERATURE

The Research on task scheduling of mixed criticality systems in recent years by S K Baruah [1] presents a hybrid system model utilizing Federated Scheduling, (i.e., each individual task is either restricted to execute on a specific processor or has exclusive access to all processors.) The Algorithm proposes an improvement to the existing deadline by introducing Mixed Criticality as a parameter. A model is proposed for mixed-criticality recurrent tasks that extend the (previously proposed) implicit-deadline sporadic DAG tasks model to account for mixed criticalities. A federated scheduling algorithm is presented and proven. A quantitative evaluation of its efficacy is derived via the widely used speedup factor metric. This model considers tasks that are allocated by Federated Scheduling while remaining tasks are partitioned amongst a pool of shared processors. Thus, the simplicity of partitioned scheduling is maintained, and capacity loss due to partitioning is capped at no larger a fraction of the platform capacity than was the case with partitioned scheduling of sequential tasks.

D. Tamas-Selicean et al [2] offer a system composed of heterogeneous processing elements (PEs) to implement hard real-time applications with different Safety Integrity levels, to include non-critical functions. It is stated that the positivity towards this approach is that the combination of hardware and software architecture allows for both spatial and temporal partitioning. A Simulated Annealing-based approach determines a task map for PEs and the sequence and length of time partitions for each PE such that applications are scheduled utilizing a Tabu Search-based approach.

This model considers a set of tasks that must be run on specific cores only in addition to three different task sets that can be run on multiple cores. The tasks that have a Specific core requirement are scheduled first, then the paper compares the performance

between three different scheduling implementations for the remaining task sets; regular mapping without partitioning, mapping with partitions, and simultaneous mapping and partitioning. A second approach results in the system going over the deadline, while the third case achieves a significant improvement in the Degree of Schedulability, showing that tasks can be scheduled on a slower or cheaper platform.

R. Medina et al [3] considers the concept of dual criticality: High Criticality (HC) and Low Criticality (LC). Two Task Graphs are created for each function and each graph is assigned HC Task Nodes. The graphs show dependencies and precedent tasks for each individual task in detail.

The Scheduling Algorithm functions on HC task deadlines. HC tasks are scheduled using the as late as possible method. When the algorithm is in LC mode, and a HC task exceeds its allotted execution time, a Timing Failure Event occurs. This switches the algorithm to HC mode, where all LC tasks are abandoned. HC tasks are scheduled first to allow ample execution time within their respective deadlines, then LC tasks are assigned in the gaps. This is a variation of Federated Scheduling from the first paper.

This Algorithm is tested with multiple cases differing the frequency of applications, the number of tasks in each application, and the number of cores. When a higher number of cores are used the system is scheduled easily but system usage is reduced below its potential.

M. Bagheri et al [4] address the under-utilization problem found in Mixed-Critical systems by proposing a scheduling method that increases overall performance but all deadlines of SC tasks are met even in the presence of transient faults with a fault tolerant scheme of task-level temporal redundancy implemented by check-pointing with rollback capabilities.

Pessimistic estimations made while designing an SC system are:

a) Worst-case execution time (WCET) used in schedule tables
b) Fault-tolerance using redundant spatial or temporal resources for the maximum number of anticipated tolerable faults

The paper also compares three different models

1) Non mixed criticality Fault tolerant where every task has a recovery slack including Safety critical and non-safety critical tasks and schedules within the runtime
2) Mixed Critical Fault Tolerant which considers redundancy and re-computation for only SC and eliminates FT schemes for NSC tasks
3) This final model has an additional runtime monitoring which adds slack to the NCS in case of a transient fault but still has FT slacks for SC scheduled always keeping in the actual time budget.

Fault Tolerant schemes use re-execution. Faulty hard processes are recovered. Faulty soft processes are recovered only if hard processes will not miss deadlines, otherwise, the soft process is dropped.

A new metric of Criticality is introduced to task scheduling by S. Vestal [5] and further surveyed by A. Burns et al [6] in their survey. HC tasks are required to be executed within the assigned deadline whereas LC tasks may be rescheduled or rearranged to accommodate a HC task. Criticality can be related to the importance of the task to the whole system function. Some tasks change criticality based on the Goal set. Some tasks can be Mission Critical when there is no collateral damage by a system failure or Safety Critical if there is a possibility of damage to surroundings of the system which is when tasks change their criticality levels.

Tasks can change criticality over time. If a task exceeds its execution deadline it triggers the system to High Criticality execution mode until all HC Tasks are executed. It then switches back to the less critical mode.

Different types of scheduling techniques are employed in MCS a few of which are:

a) Real Time Analysis: monitoring the deadlines of the critical tasks and execution times and scheduling to prevent going over the deadline.

b) Slack Scheduling: a LC task is scheduled in the slack created by HC tasks executed ahead of the deadline.

c) Period Transformation: slashing a lengthy LC task into parts and scheduling it in wherever possible in order to avoid excessive delay

d) Earliest Deadline First: scheduling all the HC tasks using the ALAP algorithm and schedule all the LC tasks in the gaps created by the HC tasks.

In multiprocessor mixed criticality systems there arises the problem of task allocations to the different processors based on the processor capability and the requirement of the critical task.

E. Azari et al [7] and S. Tosun et al [8] discuss A hardware software Co-Design environment consisting of Hardware ASIC components and Software components that function together to improve efficiency better than either only hardware or software components. Software components perform tasks utilizing less area and less energy. Hardware components execute significantly faster but consume more area and energy. Hardware components can be expensive to modify or correct after the initial design.

This research provides a way to identify tasks which when assigned to hardware components would significantly improve the design.

A hot path is tasks that are executed frequently and usually dictate total execution time. Other paths are cold paths. A Critical Path would be the path that has the longest execution latency in the design. The hot path and the critical path are mutually exclusive and can share some, all, or no, nodes.

Identifying a specific node in the hot path that decreases overall latency when assigned to a hardware component is done by the algorithm provided. Once nodes are assigned latencies are re-computed and area constraint is determined as well. S. Tosun's paper discusses also of reliability of the ASIC or the software components as well along with time and energy constraints.

Task Re-computation and Scheduling has been discussed by Giovanni De Micheli [25] Some Scheduling Algorithms discussed in Synthesis and optimization of Digital Circuits.

a) ASAP: Starts scheduling basic tasks in cycle 1 and fills in the dependent tasks after that. Tends to complete branches as soon as possible.

b) ALAP: Starts by scheduling final tasks and adjusting the predecessors after. It needs the final time constraint so that it can assign the tasks in the last possible clock cycle.

c) Hu's Algorithm: It schedules the starting tasks from a pool of tasks based on the resource constraint given.

d) LIST L: a scheduling algorithm for achieving minimum latency

e) LIST R: for achieving minimum resource usage

f) Force Directed: a new Metric Force is calculated for each task and of those available to be scheduled in the first cycle the task with the least force is scheduled first.

In P. K. Saraswat et al [10] Critical tasks, called hard tasks, are scheduled using the EDF algorithm and a Constant Bandwidth Server (CBS) to schedule soft or LC tasks. CBS enables QoS calculations for soft tasks and provides temporal isolation.

Hard real time tasks missing a deadline can lead to failure of the system. Worst case estimates (WCET) are used while scheduling and mapped in a heterogeneous architecture system. Soft task missing a deadline won't induce a failure but could affect system performance. Tasks can be completed in variable execution times. If we use WCET for soft tasks it will be a costly implementation and if we use the average execution time, there will be an increase in deadline misses.

Quality of service is the probability of completion of a soft task in the deadline given. A table with execution times of the task with each processor is given and their deadlines stated. Using a tabu search algorithm the tasks are mapped to the optimal processor.

V. Izosimov [11] states that for fault tolerant real time systems three types of faults occur. Permanent faults that require a hardware fix, transient faults which can be fixed by a fault tolerant procedure and intermittent faults which occur randomly. This paper considers both transient and intermittent faults as the same and applies corrections as such.

The fault tolerance methods proposed in the work are checkpointing, rollback recovery and active replication. Tasks are scheduled through quasi-static cyclic scheduling. Hardware replication for transient faults is costly. That leaves checkpointing recovery and re-execution. When applied in a straightforward manner the result is going over the estimated deadline. Rollback recovery and checkpointing provide time redundancy while active replication gives space redundancy. Dividing a task and having checkpoints at the end of each division identifies a fault. The part of the task where the

fault occurred is re-executed. Replication can be done in two ways. Active replication is the execution of the task simultaneously in two different processors. Backup replication runs the task on a different processor after an error is detected in the previous iteration.

S. K. Baruah et al [12] discuss the criteria to be fulfilled by a Mixed Critical System to be deemed operational. Safety critical embedded systems are required to go through certification processes. These certification requirements in mix criticality systems cause very different scheduling problems and are not satisfactorily addressed using techniques from conventional scheduling theory. This paper studies a formal model that represents such mixed criticality workloads. It also demonstrates the intractability of the system and if it could meet all its certification requirements and specifically 2 sets of certification requirements.

This paper also discusses the metric of processor speed up factor and the effectiveness of two techniques namely reservation-based scheduling and priority-based scheduling and proves that priority-based scheduling is the superior of the two.

Using the example of an unmanned aerial vehicle also known as UAV which consists of two different criticality functionalities; mission critical and flight critical. Flight critical functionalities would be certified by authorities such as the US Federal Aviation Administration. Certification authorities are not concerned with mission critical functionalities of the system which are validated by system designers separately.

C. Bolchini et al [13] talk about a design methodology that improves the system level design flow for embedded systems in order to increase reliability awareness. A mapping and scheduling algorithm are used for the application of hardening techniques to fulfill the required fault management properties that the final system must exhibit. Not all parts of the system need to be hardened against faults. The reference architecture is a

complex distributed platform with resources in terms of performance and fault detection/tolerance mechanisms

For a specific mission or safety critical application scenario dependability attributes such as reliability availability and safety are the primary concern. With technology scaling, higher frequencies and power densities the probability of components being affected negatively or being susceptible to environmental disturbances such as radiation and electromagnetic interference. This results in transient or soft errors which do not damage the hardware permanently but still affect the result.

The reliability driven system level design methodology for embedded systems is a performance optimized hardened implementation of a design space exploration process that implements a set of techniques exploiting fault management features provided by the target architecture. Key features include the following:

- Support of mixed critical management properties 4 different parts of the application
- Customizable and extendable set of fault tolerant techniques
- Voting and checking activities knew light
- Fault management features of existing processing units to be used
- Transparency of reliability design stage so the consumer can select reliability features

V. Izosimov et al [14] present an approach to fault tolerant scheduling for embedded applications with soft and hard real time constraints which guarantee the deadline for hard processes even in fault cases. Processor re-execution is employed to recover from multiple faults. A static scheduled computer is not fault tolerant. Computing a new schedule every process failure incurs inaccessible overhead. The proposal is a quasi-static scheduling strategy where a set of schedules is synthesized

offline and a runtime scheduler selects the appropriate schedule based on fault occurrence and execution time.

Faults can be differentiated into three types, permanent, transient, and intermittent. Permanent faults are mostly hardware faults and are not repaired easily, whereas transient faults and intermittent faults appear for a short time and can be dealt with by restarting the system or re-executing a task. Transient faults are the most common due to greater complexity higher frequency and smaller transistor sizes.

Every system has a timing constraint. System behavior does not depend solely on logical results but also on the time it takes to produce a result. Real time systems have hard tasks and soft tasks. When a hard task fails the result is a critical failure. When a soft task fails the result is a reduction of effectiveness.

The proposed approach integrates fault tolerance into the framework of static scheduling. Static schedules are generated beforehand and inserted into the application depending on process criticality. A mapping and scheduling algorithm for transparent re-execution is implemented on a multiprocessor system. Re-execution and active replication can be combined to develop a fault tolerant application without increasing the number of required processors or system overhead.

In M. S. Mollison et al [15] tracking the slack created by the variance between worst case execution time and actual execution time frees computational power for redistribution to less critical tasks and provides temporal isolation.

The workload of a UAV can be divided into 3 categories safety critical, mission critical, and background. Implementing a multicore mixed criticality environment would be advantageous to multiple single core systems connected via the network, which would increase cost and limit computational throughput.

Two problems with multicore systems are underutilization due to high WCET's and the need for temporal isolation of critical tasks. As the criticality rises the WCET becomes more pessimistic, which results in an idle processor once the task is complete.

This paper proposes an architecture that treats HC tasks as slack generators. Lower critical tasks are budgeted into this slack thereby utilizing the hardware more efficiently and scheduling the more demanding tasks first.

H. Li and S. Baruah [16] discuss the implementation of a mixed criticality implicit deadline for sporadic task systems on identical multiprocessor platforms is considered where inter processor migration is permitted. A theoretical analysis of both speed factor and schedulable conditions is discussed in this paper focusing on mixed criticality and scheduling algorithms.

Most algorithmic research focuses on single core processor systems. Recent HC systems are being implemented on multicore CPUs. Multicore CPUs increase complexity and sophistication resulting in the system becoming less uniform and more unpredictable allowing greater variation. This affects the certification authorities estimated execution times thereby increasing the time deadline for the system. The paper discusses transferring a scheduling algorithm application from a single core processor system to a multicore processing system efficiently.

D. D. Niz et al [17] present the impact of Economic trends in embedded systems forcing tasks of different criticality to share processors and potentially interfere with each other. Temporal isolation techniques prevent LC tasks from appropriating time intended for a HC task. These techniques can also work in reverse order, preventing a HC task from completing via symmetric enforcement. This phenomenon is known as criticality inversion. To circumvent criticality inversion, we assign priorities to tasks according to

their criticalities. This approach leads to lower utilization of the resources and affects the system negatively.

This paper proposes a new algorithm called zero slack scheduling which provides a new form of defense called asymmetric protection. The zero slack algorithm minimizes occurrences of high criticality tasks preempting low criticality tasks. When used in conjunction with a priority based preemptive scheduler algorithm, this proves to be an effective solution. Unlike zero slack scheduling, each algorithm performs slack analysis differently. This asymmetric protection negates the impact on the deadline due to criticality inversion while reducing the penalty on schedulable utilization.

In P. Penil et al [18] HC tasks with non-critical activities tend to share available hardware resources to optimize costs and reduce power consumption. The design of mixed critical systems requires the integration of design flows and tools to handle this. This paper presents a single source proposal where UML models analyze the flow of different design tools in a mixed critical context.

Integrating multiple functionalities in a shared computing architecture presents design complexity challenges due to interactions from different hardware resources. The design requires power optimization to reduce consumption, software packages with predesigned functionality to reduce design time, and heterogeneous hardware resources to minimize mean execution times.

This paper proposes a UML model that can create specific tool files to calculate the worst-case execution time while considering concurrency, different resource allocations, and multiple platform configurations. These estimates also apply static schedulability analysis tools. This infrastructure generates a cold synthesis for the deployment to run system tasks and obtain evaluations of runtime performance enabling to estimate final slack times to implement noncritical tasks.

A. Namazi et al [19] discuss a reliability aware hard real time task scheduling method for multicore systems with a quantitative reliability model. The model uses clustered replication to attain the desired reliability threshold by achieving minimum replication overhead and a latency increase also considering single and multiple soft errors.

In recent generations, multi core platforms have both bandwidth and scalability issues. The trend shifted to using the network on chip architectures to overcome these issues as the core count increased. Two major constraints for today's digital systems are real time performance and reliability. The digital systems of today must satisfy hard real time constraints by achieving both temporal and logical correctness in their results. Susceptibility to errors has increased due to continuous transistor scaling. Different criticality levels introduce the requirement for different levels of reliability systems.

Task mapping holds the key to these crucial issues as it specifies which task should run on which core in the system. The proposed reliability aware task scheduling on NoC based platform uses modified clustered replication with the majority voting to achieve reliability. A multi-step heuristic algorithm can drastically reduce time to find a probable task mapping solution for hard real time applications while reducing the replication overhead to maintain a reliability threshold. the proposed method schedules hard real time tasks with minimum redundancy overhead and better communication overhead in comparison with the conventional replication method also giving a bonus execution latency in the simple re-execution method.

R. I. Davis et al [20] talks about the problems of priority assignment in multiprocessor real time systems using global fixed priority preemptive scheduling. It demonstrates that an optimal priority assignment algorithm usually designed for single

processor scheduling can also be applied to the multiprocessor case upon satisfying three conditions with respect to schedulability tests.

Multiprocessor real time scheduling can be divided into two segments. A global partitioned approach allocates each task to one processor approaching it like single processor scheduling. The global priority approach switches tasks from one processor to another during runtime. These scheduling algorithms can be divided into 3 types, fixed task priority, fixed job priority, and dynamic priority. The paper discusses priority assignment policies for global fixed task priority preemptive scheduling referred to as global FP scheduling.

J. Theis et al [21] adapts the research for event triggered systems to time triggered approach which has better certification feasibility and compares their resource utilization guarantees for TT systems. Adding mixed critical scheduling to legacy TT systems while leaving the existing schedule unchanged adds a simple change to adapt to criticality. Although TT systems are favored by certification authorities, they ensure noninterference by strict isolation between components, thus causing low resource utilization.

The proposed model studies the existing schedule table, run a simple online execution, and emulates criticality change. Changes are suggested only if required by the system, thereby reducing effort. If changes are made the system must be recertified but this is not a frequent occurrence. The algorithm analyses the runtimes and slack which it uses to provide flexibility for the system.

A federated scheduling algorithm MCFQ is presented by R.M. Pathan [22]. The feature of this algorithm is having alternative schedules computed to assign each HC task to the processors. The algorithm carefully selects the schedule which enables all other tasks to be scheduled on remaining processors. This method has a higher likelihood to satisfy the total resource requirement. Slack generated by the selection minimizes the

total resource utilization by the tasks and can be used to improve system QoS. The parallel programming paradigm enables to utilize the processing capabilities of the multi core architecture. Thereby seeing each parallel task as an individual DAG.

The algorithm calculates nominal and overload values for total work and length of every task. Each task is assigned a virtual and critical deadline. In the beginning, tasks run in virtual deadlines. The task with the highest overload value is assigned to dedicated processors while the residual tasks are scheduled on the remaining processors. The system can run in either typical or critical state, and switches between them when a task isn't complete by end of its virtual deadline. Low tasks are dumped to allocate processing power to high tasks. By maximizing the number of LC tasks not discarded in critical mode the QoS of the system is improved.

M. Hassan [23] discusses the challenges and research opportunities in the combination of Mixed critical systems with the Multiprocessor system on chip architecture. The proposed model gives flexibility to scheduling a HC task to a higher order PE to negate the execution latency. The criticality is not restricted to two or three levels, as recent studies are considering as many as five to six levels of assurance in tasks. MPSoC's provide cost, area, power, and performance improvements to the MC system. The paper focuses on four parameters, theoretical model, timing interface, data sharing, and security.

The challenges of the theoretical model to consider are as follows:

- Switching constraints due to heterogeneity of SoC's
- Scalability and switching overheads
- Worst case ET's
- Timing interface
- Many of PE's

- Different memory for each PE
- Can’t have tight bounds.

Implementing a mixed critical system single ASICs and programmable processors is discussed by Kalavade et al [24]. Custom hardware is used to implement the intense portions of the system and the rest are implemented by software. This allows the system to meet performance requirements with reduced design costs. Using only application specific integrated circuits increases the performance of the system but is a very costly approach in mixed criticality systems, so a hybrid approach is used to implement the system called the hardware software co-design.

The problem with this model is managing its four processing stages; partitioning, synthesis, co-simulation, and design methodology. In a DAG each node has four parameters; area, code size, hardware ET and Software ET. The partitioning problem is to find a mapping of nodes to hardware and software taking into consideration the communication between the nodes and keeping the area occupied by nodes mapped to hardware to a minimum.

In B. Nimer et al [9] the Freedom of each task is calculated and is used to determine the priority of competing tasks to schedule to earlier control steps. Tasks with lower freedom value get priority in scheduling. This will result in the initial and most reliable schedule.

If the resulting schedule exceeds the latency deadline, performance optimization techniques are applied. To decrease overall latency, tasks that can be scheduled concurrently are identified and assigned to different PEs without violating dependency conditions. The task with the highest delay is assigned to a faster and more reliable PE among other candidates. If both tasks have the same latency value, we choose the one with the lowest criticality value and assign it to the next PE.

If the deadline is still not achieved, slower tasks in the critical path are iteratively assigned to faster PEs starting with the least critical tasks and repeated until the performance deadline is met. If area constraint is not met, tasks are iteratively re-assigned with higher freedom values going to a candidate PE with the highest reliability among others. If this approach is determined to increase latency upon calling the original scheduling algorithm again, a less reliable PE is assigned. If none of the PEs work, tasks are reassigned to slower ASICs with lower area costs until area bound is met.

Tasks are re-computed on idle processors for later tasks instead of storing data when done first to reduce memory time.

Theis et al [26] talk about Mixed critical systems are usually event triggered but many safety critical domains favor a time triggered approach. This paper presents an effective flexible approach for transition of mode change before time triggered systems in mixed criticality jobs. The time triggered application in which all the activities of the system are triggered by time progression only so that a schedule for the entire duration of the system is drawn before runtime. the decisions made are determined by the precomputed schedule also known as the schedule table. This table is easier to verify hence more popular in certification authorities.

However, in mixed criticality systems, the inflexibility of time triggered approach has the drawback when it comes to the task with different assumptions that can't fit in a single table. The research conducted for even triggered systems can be applied for time triggered systems as well such as having two different schedule tables one based upon the system designer and gather according to the certification authority parameters. The system starts with the system designer schedule table end proceeds to run along with it. When there is a fault, or a task goes beyond the execution time during runtime a switch

happens, and the table is switched with the certification parameters and proceeds to run till the end with the same parameters.

While building two matching schedule tables we need to consider feasible and consistent switching during runtime and transferring the computational requirements for the ongoing jobs. Since these have two different execution times one for each criticality level the standard algorithm cannot be applied directly. The algorithm proposed succeeds in transferring which criticality principles the time triggered domain, time triggered framework designed for Mixed criticality systems, generating the schedule tables needed by this framework also. The algorithm constructs schedule tables for runtime execution is far more efficient resource utilization and improved flexibility.

A Namazi et al [27] proposes new reliability centered task mapping approach in a multi-core platform at design time for applications with DAG-based task graphs. The main goal is to devise a task mapping algorithm that meets a target reliability threshold at the cost of a controlled performance degradation. The proposed approach uses a majority-voting replication technique to achieve error-masking capability.

Zhang et al [28] introduce a novel swarm intelligence optimization algorithm called the firework algorithm (FWA) and applies it to hardware/software partitioning. In Yao et al [29] a mixed-criticality sporadic task model with multiple virtual deadlines is built and a certification-cognizant dynamic scheduling approach referred to as the earliest virtual-deadline first with mixed-criticality (EVDF-MC) is considered, which exploits different relative deadlines of tasks in different criticality modes. Wang et al [30] propose a multi-criticality graph-based end-to-end (MCE2E) task model. This task model is abstracted from the fault diagnosing and fixing the process of the industrial control systems. The task in this model consists of a collection of nodes representing mixed

criticality modes. Each node is defined by a parallel directed acyclic graph, the subtasks of which are pre-allocated to multiprocessors.

R. Trub et al [31] develop a mixed-criticality runtime environment on the Kalray MPPA-256 Andey many-core platform. The runtime environment implements a scheduling policy based on adaptive temporal partitioning. They develop models, methods and implementation principles to implement the necessary scheduling primitives, to achieve high platform utilization and to perform a compositional worst-case execution time analysis. A. Thekkilakattil et al [32] 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 Zonal Hazard Analysis (ZHA) and Fault Hazard Analysis (FHA). L. Sigrist et al [33] discusses how to combine this policy with an optimization method for the partitioning of tasks to cores as well as the static mapping of memory blocks, i.e., task data and communication buffers, to the banks of shared memory architecture.

L. Sha et al [34] discusses how mode changes can be accommodated within a given framework of priority driven real-time scheduling. R. Schneider et al [35] present a multi-layered schedule synthesis scheme for MCCPS that aims to jointly schedule deadline-critical and QoC-critical tasks at different scheduling layers. Y. Zhou et al [36] propose a design framework comprising a hyper-period optimization algorithm, which reduces the size of the schedule table and preserves schedulability, and a re-scheduling algorithm to reduce the number of preemptions.

L. Zeng et al [37] presents design methodologies to guarantee both safety and schedulability for real-time mixed-criticality systems on identical multicores. Assuming hardware/software transient errors, model safety requirements on different criticality levels explicitly according to safety standards; based on this, they further propose fault-

tolerant mixed-criticality scheduling techniques with task replication and re-execution to enhance system safety. R. M. Pathan et al [38] comes up with an effective scheduling policy and its analysis that can guarantee certification of the system at each criticality level while maximizing the utilization of the processors. D. Müller et al [39] review EDF-VD's schedulability criteria and determine its schedulability region to better understand and design mixed-criticality systems. S. Maurer et al [40] present a generic component and communication model for CPS that not only allows the co-existence of computing paradigms of different criticality but also supports the data exchange between them.

J. Lin et al [41] study a problem of scheduling real-time, mixed-criticality tasks with fault tolerance. An off-line algorithm is proposed to enhance the performance of the system when it runs into a high criticality mode from a low-criticality mode. A novel on-line slack-reclaiming algorithm is also proposed to recover from as many faults as possible before the jobs' deadline.

Z. Li et al [42] proposes a two-phase execution model is proposed for mixed-criticality (MC) tasks. Different from traditional MC tasks with a computation phase only, the two-phase execution model requires a memory-access phase first to fetch the instructions and data, and then computation. Theoretical foundations are first established for a schedulability test under given memory-access and computation priority assignment. Based on the established theoretical conclusions, a two-stage priority assignment algorithm, which can find the best priority assignment for both memory-access and computation phases under fixed-priority scheduling, is further developed. V. Legout et al [43] approach exploit the ability of tasks with low-criticality levels to cope with deadline misses. On multiprocessor systems, our scheduling algorithm handles tasks with high-criticality levels such that no deadline is missed. For tasks with low-criticality

levels, it finds an appropriate trade-off between the number of missed deadlines and their energy consumption.

J. Lee et al [44] propose a new scheduling algorithm and develop its runtime schedulability analysis capable of capturing the dynamic system state. Our proposed algorithm adaptively determines which task to drop based on the runtime analysis. To determine the quality of task dropping solution, we propose the speedup factor for task dropping while the conventional use of the speedup factor only evaluates MC scheduling algorithms in terms of the worst-case schedulability.

K. Lakshmanan et al [45] present a ductility-maximization packing algorithm to complement our previous work on mixed-criticality uniprocessor scheduling. Our packing algorithm, known as Compress-on-Overload Packing (COP) is a criticality-aware greedy bin-packing algorithm that maximizes the tolerance of high-criticality tasks to overloads.

P. Huang et al [46] model explicitly the safety requirements for tasks of different criticalities according to safety standards, assuming hardware transient faults. We further provide analysis techniques to bound the effects of task killing and service degradation on system safety and schedulability. N. Guan et al [47] present an algorithm called PLRS to schedule certifiable mixed-criticality sporadic task systems. PLRS uses fixed-job-priority scheduling and assigns job priorities by exploring and balancing the asymmetric effects of the workload on different criticality levels. D. Socci et al [48] present a state-of-the-art STTM algorithm that works optimally on a single core and shows good preliminary results for multi-cores. D. Socci et al [49] propose an algorithm that is proved to dominate OCBP, a state-of-the-art algorithm for this problem that is optimal over fixed job priority algorithms. J. Ren et al [50] present a partitioned scheduling scheme for mixed-criticality tasks on multiprocessor platforms that address both issues. Our

scheduling scheme consists of (i) a task-to-processor packing algorithm that takes into account the demands of tasks with respect to their criticality levels, and (ii) a mixed-criticality uniprocessor scheduling strategy that is based on task grouping.

CHAPTER III:

SYSTEM MODELLING

3.1 Target Architectural Model

A Processor comprises an electronic circuit that performs operations on data and provides a required output. Processors have developed through history and evolved into multiple variations. Predominantly among those are two types of processing units namely hardware processors and software processors. Hardware processors are built to perform a preset operation on the data and are not easily modified whereas a software processor is a user programmable device that can be molded according to the requirements.

Some common type of hardware processors is Application Specific Integrated Circuits or ASIC's. ASICs are modeled according to a pre-defined requirement and designed using a hardware description language (HDL) accordingly. Once defined and implemented it is not a simple process to alter the functionality of an ASIC. These are produced on a huge quantity for a particular use. Microprocessors are a type of ASIC's.

The most common form of Software Processors are Central Processing Units or CPUs. These form the basic block of every modern-day computer. CPU's are programmed to follow a set of instructions given by the user. These instructions can be modified according to the requirements and the CPU can be easily reprogrammed to execute a different operation with each set of instructions.

ASIC's are best utilized in environments that require a task to be repeated over and over in a little span of time. ASIC's fulfill that criterion by being able to perform operations very quickly compared to a software processor and more efficiently. A downside of using ASICs is the huge power requirement, expensive design cost and lack the freedom to rectify the chip after design.

CPU’s are flexible, reprogrammable and utilize less space. Which makes them the perfect choice for environments where the requirements change frequently. They perform the required operation reliably and can modify operation with just a change of instructions. CPU’s are more user friendly and easy to design but takes time to process the instructions and risk missing the deadlines for systems.

Current day demands have outgrown the utilization of a single processor and are requiring a multi core architecture to implement their functions. This arises 22 types of architectures namely homogeneous and heterogeneous architectures. Homogeneous architectures are basically formed with the same type of processors, either hardware processors our software processors. Having an architecture solely dependent on hardware processor arises the problem of expensive design costs and large space requirements. On the other hand, using only software processors would risk the system to go beyond the time constraint and frequent failure to meet deadlines.

These drawbacks introduced the need for heterogeneous architectures where a combination of CPUs and ASICs to be used in conjunction with each other in order to reach the expectations of the modern-day requirements. A heterogeneous architecture would negate most of the individual negatives by assigning repetitive tasks to faster GPU's end the complicated tasks to the CPUs thereby creating an environment that is immune to the individual fallbacks off the processing components. A few examples of heterogeneous architectures in real life would be microwave ovens telephone applications etc.

In the current thesis, we use a system which utilizes 2 CPUs and 2 ASIC’s to process the given tasks according to the algorithms discussed later in the document. This provides the flexibility to switch between hardware and software components according to the requirement of that instant. The CPUs have a longer latency period compared to the

ASIC's but offer lesser power consumption and reduction in design space requirements. The ASICs help process repeating tasks at a faster rate thereby decreasing the overall time overheads.

Depicted below is the model of architecture utilized for the experiments in this thesis.

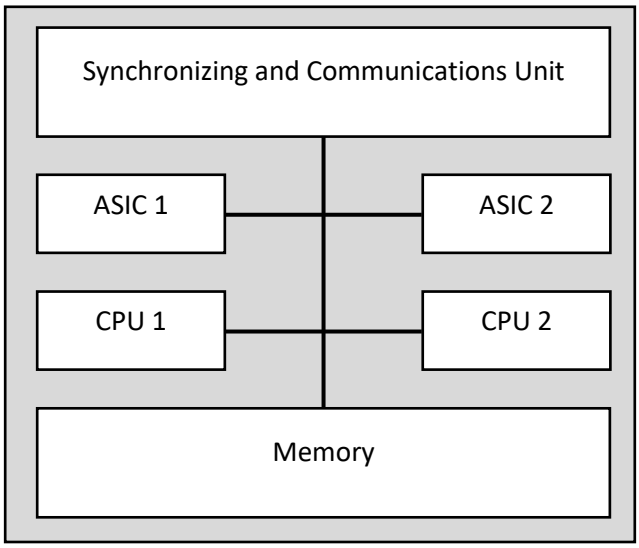


Figure 3.1 Target Architectural Model

3.2 Task Graph

A task graph is a graphical representation of a system utilizing nodes or vertices and lines traveling between these vertices. each vertex represents a task in the system and the closer to the source it is the faster it arrives. This graph depicts the flow of data from the source to the sink through the network of vertices and each connecting edge representing the dependency with its predecessor or successor within the network.

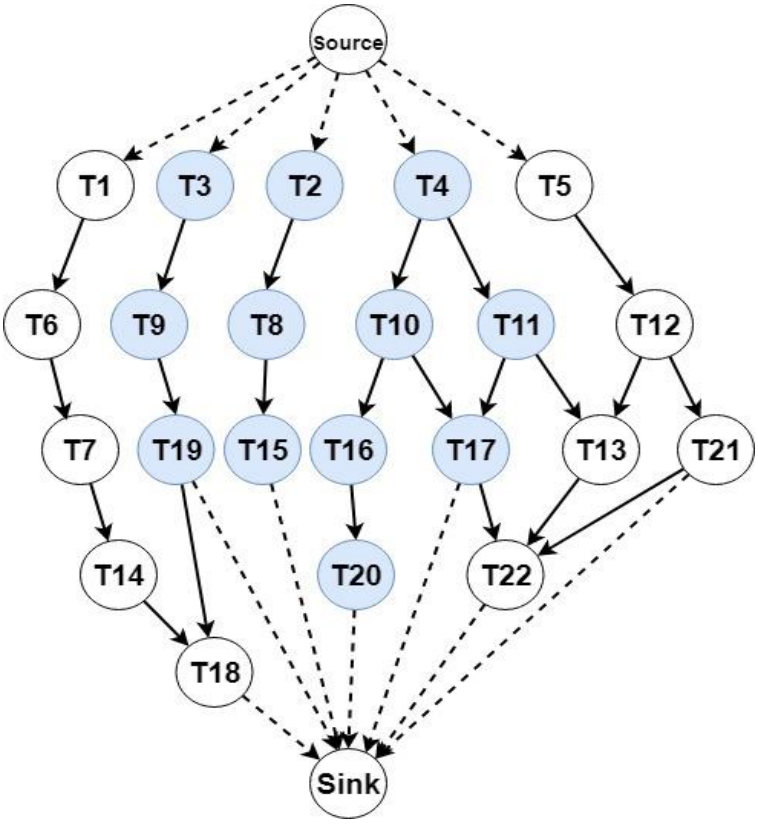


Figure 3.2 Task Graph

In Fig 3.2 The interconnections between tasks in a task graph represent the data dependency between them. This data dependency implies that without the predecessor of a certain task being executed it can't be available for scheduling in the system. Each task cannot occur without it's preceding task being executed. If there are multiple predecessors for a certain task it won't be available for Scheduling until and unless all its predecessors have been scheduled and executed. A task can have multiple predecessors and multiple successors. The same theory applies to the successors of a task as well.

The task graph used in this thesis does not have loops or decision blocks. They represent the data flow between the tasks from the source and till the sink. we state the fact that if a task is deemed critical all its predecessors would be considered critical tasks

as well. This fact is based on the logic that if a predecessor is not considered critical then the system risks failure in the task by not executing its predecessor in time.

3.3 Mixed Criticality Systems

We define a system as a set of tasks interconnected with each other and working together towards producing an output. There can be many types of systems simple, complex, hybrid, etc. Each system has a different set of tasks and each task different from each other. Tasks of a system are arranged according to a priority that facilitates a smoother scheduling process. These priorities vary with the requirements, such as the earliest arriving tasks. Tasks with a longer path to completion. A more important priority assignment in contemporary usage is Criticality.

The criticality of a task defines the impact of its execution on the outcome or the status of the system itself. Criticality also helps determine the importance of the task in the system. A Critical task example would be, brakes in a Vehicle, without these the safety of the car will be compromised and poses a risk to the users operating the car. Critical tasks are further classified into Safety-Critical tasks and Mission Critical tasks.

Safety-Critical tasks are the set of tasks that ensure the safe operation of the system and its surroundings in all modes of operation. Mission-critical tasks ensure the proper execution of the basic system function with no failures. To illustrate the difference between these tasks, consider a surveillance drone. The tasks that work to keep the drone's navigation and control are categorized under Safety-Critical so that in times of an emergency the drone can make a controlled exit and not cause damage to itself or its surroundings. The surveillance and data transmission of the system is mission-critical, ensuring that the data is collected and transmitted, fulfilling the purpose of the drone. In case of an emergency, all these tasks must work to operate the drone.

Every system has some critical tasks and non-critical tasks that contribute to the success of the system. Execution of critical tasks ensures the system works and produces and output while non-critical task execution improves the quality of the result obtained. This combined system of critical tasks and non-critical tasks is called a mixed critical system. A common mixed critical system is commercial aircraft. In an Aircraft, the system that works to navigate and keep the craft airborne is considered critical than the inflight entertainment. In times of an emergency, the priority would be to keep the Critical tasks running and the non-critical tasks would be dropped as they can't affect or help improve the operation of the system.

Mixed critical systems have further divisions based on the criticality levels, Dual Critical and multi critical, etc. These systems drop LC tasks in case of an emergency. This although doesn't affect the working of the system but reduces the quality of the output produced. Algorithms that consider the non-critical tasks to an extent in scheduling are designed to deal with this drawback.

3.3.1 High Critical and Low Critical Tasks

We consider a Dual Critical system where the tasks are divided into two types of Criticality, High critical (HC) tasks, and Low Critical (LC) tasks. HC tasks are given priority during scheduling over LC tasks. HC tasks are essential to the survival of the system whereas the LC tasks improve the effectiveness of the system. When scheduling the tasks, the algorithm has to schedule all the tasks within the time constraint and may ignore LC tasks for a HC task.

We state the fact that a High Critical task cannot have a LC task as a predecessor thereby declaring all the predecessors of a HC task to be assigned HC priority. This is

due to the problem that if a HC task depends on a LC task for data which may be ignored in some cases subsequently affecting the HC task risking the whole system.

In Fig 3.3 A depiction of the task graph is presented where each node represents a task and each line represents the data flow and dependencies between each task. HC tasks are filled in color.

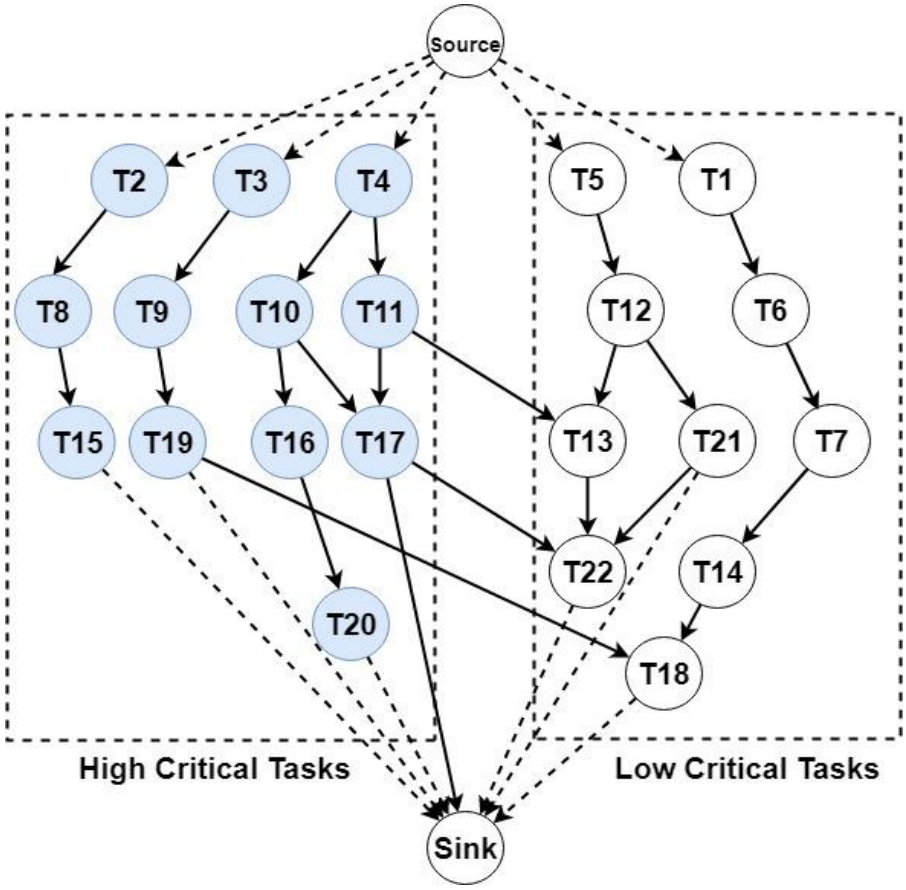


Figure 3.3 Criticality of tasks distinguished

3.3.2 Criticality Modes of the System

The system discussed in this thesis operates in 2 different modes namely High criticality mode and low criticality mode. Low criticality mode considers all the tasks to be of equal priority and work towards scheduling all the tasks in time without dropping or

interrupting any task. low criticality mode can also be considered as the normal operating mode for the system. The system transitions into high criticality mode under certain conditions predefined by the design. Having a HC task error while executing in low criticality mode, and approaching emergency, manual override are a few ways of triggering high criticality mode.

When the system is in high criticality mode it ignores all the LC tasks for the time being and schedules the HC tasks according to the priority assignment. After all the HC tasks have been scheduled the LC tasks are scheduled in the slack generated between and after the scheduling of HC tasks. If in the case of any LC task going beyond the desired deadline it can be dropped and the system can proceed into the next cycle of execution. The design tries to accommodate as many LC tasks as possible into the schedule in order to improve the overall efficiency of the system simultaneously ensuring the safety of the system.

The task graph can be reimagined into the following way to provide a clearer understanding of the about statements all the HC tasks are arranged to the left and all the LC tasks are arranged to the right of the graph. The dependencies between these 2 categories only extend from HC tasks to LC tasks and not the other way around.

3.4 Constraints

A perfect system without failure is impossible to achieve. So, the system is assigned some constraints that would deem the system viable in a real time environment. These constraints define the system limitations and set a target to achieve utilizing limited resources.

Reliability constraint is the target reliability the system must achieve in order to be put into practical usage. Reliability is essential to systems running in populated environments to ensure the safety of the system

Latency constraints are set to achieve the maximum reliability possible while executing the system within a time bound. Latency is prioritized when the system is under threat and must transfer data collected so far.

The system can have many constraints to consider at a time. In this case, we try to achieve the highest possible reliability by utilizing all the processors optimally. While a second scenario the system is executed in an as fast as possible method.

3.5 Reliability Modelling

Reliability is defined by how successful the task can run on a specific processor. The Reliability of each task is different on each of the processors in the architecture. the total reliability of a system with multiple units is always the product of each individual task Reliability in the system. As in a case of two numbers, both lesser than 1 their product is always lesser than each multiplicand. This phenomenon also shows up in system reliability as any task reliability is always less than 1, with 1 being perfect execution the overall Reliability is always lower than any individual task Reliability.

$$R_{overall} = \prod_{i=1}^{n} (R_i)$$

To improve the reliability of the system the individual Reliability of the tasks must be improved. This is achieved by adding a redundant component to each task. This can be accomplished in two types of configurations, series, and parallel.

A series configuration affects the overall reliability in a negative way. The reliability of a component multiplies with the reliability of every other component in the series and reduces the overall reliability less than the individual reliability of each component. With series redundancy, lower reliability is a compounding problem. The formula for the series reliability is below.

$$R_{overall} = \prod_{i=1}^{n} R_i$$

When we configure redundancy in parallel, the components are connected such that the input divides into all the components and the outputs of the components combine into one, thereby increasing overall reliability. The formula for parallel reliability is below.

$$R_{parallel} = 1 - \prod_{i=1}^{n} (1 - R_i)$$

3.6 Technology Library

The performance details of the tasks on each of the processors are tabulated into a tech library. This would contain the data required for the algorithm to make decisions. In this case, the tech library consists of the Reliability of each task on each of the processors and the time taken to complete the execution. This will later be utilized by the algorithm to decide which processor is the best to achieve the best Reliability or faster latency.

Table 3.1
Technology Library

		T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17	T18	T19	T20	T21	T22
ASIC 1	**Reliability**	0.995	0.996	0.998	0.992	0.991	0.994	0.979	0.970	0.986	0.992	0.998	0.969	0.989	0.979	0.978	0.998	0.974	0.976	0.990	0.997	0.997	0.995
	Delay	8	10	12	6	7	9	10	15	5	13	16	12	12	15	6	12	9	6	12	15	14	10
ASIC 2	**Reliability**	0.997	0.998	0.994	0.995	0.985	0.979	0.993	0.996	0.988	0.955	0.991	0.987	0.993	0.971	0.990	0.991	0.971	0.998	0.993	0.995	0.986	0.993
	Delay	10	12	9	8	10	11	12	12	7	17	12	7	10	10	12	8	9	8	8	12	8	11
CPU 1	**Reliability**	0.994	0.994	0.986	0.987	0.989	0.969	0.995	0.993	0.972	0.989	0.985	0.990	0.987	0.973	0.985	0.987	0.973	0.989	0.980	0.996	0.993	0.984
	Delay	50	40	35	45	35	50	25	25	35	30	50	30	35	20	55	55	30	25	65	40	25	30
CPU 2	**Reliability**	0.993	0.980	0.980	0.977	0.975	0.976	0.999	0.995	0.980	0.985	0.990	0.978	0.996	0.991	0.987	0.983	0.970	0.985	0.990	0.995	0.987	0.995
	Delay	55	45	30	65	55	35	35	50	65	35	30	45	25	40	50	60	35	30	50	35	45	50

CHAPTER IV:

MOTIVATIONAL EXAMPLE

To illustrate our approach, we work with the given task graph and associated technological library in Table 3.1 above. The sample task graph in Fig 3.1 contains 22 task nodes to be executed in from source to sink. The technological library in provides the reliability and execution time for each tasks on each of the processors. The algorithm is designed to tackle shortage of resources. So, the task graphs we use as an example are configured such that there are more tasks than processors available throughout the runtime to illustrate the algorithms effectiveness in such scenarios.

The algorithm runs in multiple iterations, in each iteration selecting the task with the highest priority and assigning to the appropriate processor available. The priority conditions can be configured ahead based on the user requirement. In this case the different parameters of each task such as longest path to sink, etc are looked in to determine the priority. In case of a task has no idle processor to be executed upon the algorithm waits till the time one of the current tasks completes its execution. As tasks become available to be scheduled, task priority is recalculated at the start of each iteration and the current top priority task is chosen to be scheduled over a previously higher priority task which wasn't scheduled due to lack of idle processor.

Two Algorithms operating in two different criticality modes of the system are designed for each of the two desired results. First of which is the reliability priority approach which prioritizes task reliability during task assignment. By prioritizing each tasks reliability, the overall system reliability is bound to improve. In LC mode of the system the task priority order of the algorithm uses task criticality as a last tie breaker in case of all other parameter matching up for the tasks in contention. This is designed so

that the algorithm schedules each task irrespective of its criticality. In HC mode the algorithm works the iterations only on the HC tasks in the first cycle following a slightly modified priority assignments. After scheduling all the HC tasks onto the processors, the LC tasks are then put through the iterations to complete the system scheduling.

The same approach is also utilized in a latency priority algorithm with the intention of executing the tasks at a higher rate while watching the overall system reliability. A noticeable improvement in latency is achieved with a corresponding decrease in reliability.

After taking in the inputs from the tech library. We instruct the algorithm to list out all the tasks available for scheduling into a set T. the algorithm will choose the task with the greatest number of successors to schedule. This decision is made because the distance from each task to the sink cannot be determined until all the tasks are scheduled. So, the decision to choose the task with the number of successors such that the next tasks in line would be available earlier and the longer path getting struck in slower processors is avoided. Once the first tasks are scheduled the algorithm updates its set of available tasks as new tasks arrive and compare them with their priority. The priority of tasks in case of an equal number of successors is determined by the following conditions in that order respectively. The task with higher criticality is preferred first. The task with a greater number of HC task successors is preferred first. The task nearest to the source is preferred first. If there is still a tie the task that can run with the best latency in the available processor is chosen. This order is devised keeping in mind the survival of the system by preferring criticality and attaining better execution latency.

When the system switches into High Criticality mode the LC tasks are not considered for the scheduling in the first cycle. All the HC tasks are scheduled first following the order with the task having the greatest number of successors scheduled

first. After all the HC tasks are scheduled the algorithm now processes the LC tasks and completes the system.

The results obtained after putting the sample task graph through the algorithms discussed are depicted in table 4.1 below.

Table 4.1
Results of Motivational Example

Properties	**Reliability Priority Scheduling**				**Latency Priority Scheduling**			
	LC mode		**HC mode**		**LC mode**		**HC mode**	
	Overall	**Only HC tasks**	**Overall**	**Only HC tasks**	**Overall**	**Only HC tasks**	**Overall**	**Only HC tasks**
Reliability	0.786	0.854	0.831	0.897	0.743	0.837	0.776	0.887
Latency	110	107	162	73	107	79	97	71

The results show that the algorithms succeed in creating an improvement in reliability while operating in HC mode of the system and also for the overall system at a cost of increased latency. This is a fixable tradeoff as LC tasks can be dropped, this latency overrun can be rectified. When comparing the HC task execution in both the algorithms, low criticality mode has lower reliability than high criticality mode. This proves the success of high criticality mode's ability to increase the reliability of HC tasks

Observing the latency priority approach there is a significant improvement in latency without sacrificing much the reliability of the system. This tradeoff is utilized in scenarios where all the data collected by the system has to be transmitted as soon as possible due an approaching emergency or imminent destruction of the system.

The next sections discuss the algorithms in detail and the results obtained from each algorithm by referencing the generated schedules given in figure 6.1.

CHAPTER V:

LATENCY CONSTRAINED RELIABILITY IMPROVEMENT

Improving the reliability of the system is the primary focus of the algorithms in this approach. Algorithm 1 operates in the LC mode of the system and prioritizes reliability in scheduling the tasks to the available processors. Whereas algorithm 2 operates in HC mode processing HC tasks ahead of LC tasks thereby providing access to better reliable processors. Improvement of HC task reliability is prioritized over LC task Execution.

5.1 Low Criticality System Mode

Tech Library provides the algorithm with the reliability (R_{ji}) and latency (L_{ji}) values where 'j' refers to processor index and 'i' denotes the task number from the task graph. The algorithm now forms the set '$I_{current}$' which has all the tasks that are available to be scheduled and a set '$J_{current}$' with the idle processors. These sets are updated at the start of each iteration. A subset of 'I' is created with the tasks with the longest path to sink from '$I_{current}$'. The set I is checked if it has multiple elements and put through further priority filters until it has only one task. a few of the priority filters consider better latency on the processors available, shortest from source, better reliability. the task that passes through all the filters is scheduled onto its ideal processor. This process is repeated until all the tasks in the task graph are scheduled that is $I_{current}$ is empty and has no further tasks.

Algorithm 1 takes the input data. In steps 1 to 4, it compiles the data and arranges tasks in the order of their predecessors. From steps 5 to 18 the algorithm selects the task that has the highest priority and assigns it to the best reliable processor, the rest of the algorithm loops back to select the next task to be scheduled.

Algorithm 1: Scheduling with Low Criticality System Mode

INPUT: Task graph with 'n' tasks, Tech Library with (R_{ji}, L_{ji}) where j ∈ J (1,4) refers to processors and i ∈ I (1, n) denotes the tasks in task graph,

OUTPUT: Schedule of LC reliability priority mode, L_o, R_o, L_{hc}, R_{hc}

1. Update $I_{current}$ = {i | i are tasks that are available to be scheduled, i ∈ I}
2. Update $J_{current}$ = {j | j are processors that are idle, j ∈ J}
3. Form the set of 'I′ where
4. I = {i | i are the tasks with the greatest number of successors, i ∈ $I_{current}$}
5. **if** |I| = 1
6. **then** $T_{current} = T_i$
7. **else** I′ = I
8. I = {i | i is High Critical task, i ∈ I′}
9. **if** |I| = 1
10. **then** $T_{current} = T_i$
11. **else** I′ = I
12. I = {i | i is nearest to the source, i ∈ I′}
13. **if** |I| = 1
14. **then** $T_{current} = T_i$
15. **else** I′ = I
16. I = {i | i has greatest $R_j(i)$, i ∈ I′ & j ∈ $J_{current}$}
17. **if** |I| = 1
18. **then** $T_{current} = T_i$
19. Assign $T_{current}$ to j ∋ $R_j(i)$ is the highest, j ∈ $J_{current}$
20. **if** | $J_{current}$ | = 0
21. **then** wait for time T
22. Update $I_{current}$ & $J_{current}$
23. **if** | $I_{current}$ | = 0
24. **then** wait for time T
25. Update $I_{current}$ & $J_{current}$
26. Repeat step 4 until | I | = 0 i.e. all the tasks are scheduled

5.2 High Criticality System Mode

In Algorithm 2, the first four steps separate the tasks as HC and LC. As the steps 1 to 4 brings out all the HC tasks into set $H_{current}$. Then, steps 5 to 15 sorts the HC tasks based on priority and assigns the task on top to its ideal processor steps 16 to 22 loops back until all the HC tasks are scheduled. The steps form 23 till the end schedules the rest of the LC tasks in the same manner of priority order as the previous cycle.

Algorithm 2: Scheduling with High Criticality System Mode

INPUT: Task graph with 'n' number of tasks, Tech Library with (R_{ji}, L_{ji}) where j ∈ J (1,4) refers to processors and I ∈ I (1, n) denotes the tasks in task graph,

OUTPUT: Schedule of Low Criticality reliability priority mode, L_o, R_o, L_{hc}, R_{hc}

1. Update $H_{current}$ = {h | h are High Critical tasks that are available to be scheduled, h ∈ I}
2. Update $J_{current}$ = {j | j are processors that are idle, j ∈ J}
3. Form a set of 'H' where
4. H = {h | h are the tasks with the greatest number of successors, h ∈ $H_{current}$}
5. **if** |H| = 1
6. **then** $T_{current} = T_h$
7. **else** H′ = H
8. H = {h | h is nearest to the source, h ∈ H′}
9. **if** |H| = 1
10. **then** $T_{current} = T_h$
11. **else** H′ = H
12. H = {h | h has greatest $R_j(h)$, h ∈ H′ & j ∈ $J_{current}$}
13. **if** |H| = 1
14. **then** $T_{current} = T_h$
15. Assign $T_{current}$ to j ∋ $R_j(h)$ is the highest, j ∈ $J_{current}$
16. **if** | $J_{current}$ | = 0
17. **then** wait for time T
18. Update $H_{current}$ & $J_{current}$
19. **if** | $H_{current}$ | = 0
20. **then** wait for time T
21. Update $H_{current}$ & $J_{current}$
22. Repeat step 3 until | H | = 0 i.e. all the High Critical tasks are scheduled
23. Form a set of tasks $L_{current}$ where
24. $L_{current}$ = {l | l are the remaining tasks that are available to be scheduled with greatest number of successors, l ∈ I}
25. **if** |L| = 1
26. **then** $T_{current} = T_l$
27. **else** L′ = L
28. L = {l | l is nearest to the source, l ∈ L′}
29. **if** |L| = 1
30. **then** $T_{current} = T_l$
31. **else** L′ = L
32. L = {l | l has greatest $R_j(l)$, l ∈ L′ & j ∈ $J_{current}$}
33. **if** |L| = 1
34. **then** $T_{current} = T_l$
35. Assign $T_{current}$ to j ∋ $R_j(l)$ is the highest, j ∈ $J_{current}$
36. **if** | $J_{current}$ | = 0

37. **then** wait for time T
38. Update $L_{current}$ & $J_{current}$
39. **if** | $L_{current}$ | = 0
40. **then** wait for time T
41. Update $L_{current}$ & $J_{current}$
42. Repeat step 23 until | L | = 0 i.e. all the tasks are scheduled

After executing above algorithms until all the tasks in the motivational example are scheduled the resulting schedule is presented in Fig 5.1.a, 5.1.b. The schedule of low criticality mode algorithm has the tasks scattered around the timeline irrespective of criticality which obstructs the availability of better processors to the HC tasks. This although does not pose any serious issues in normal operation modes it does affect the system effectiveness in critical environments. This is corrected by making the system run in two different operating modes. A low criticality mode to accommodate all the tasks equally and a HC mode where the sole aim of the system is to improve the reliability of HC tasks.

Looking at the scheduled tasks the HC mode of the system will follow the same order of priority with a small change in operation. The algorithm sorts out the HC tasks from the pool of tasks and assign the HC task with the greatest number of successors to be scheduled first following the same order from the low criticality mode to determine tiebreakers. After all the HC tasks are scheduled the algorithm moves on to schedule the LC tasks. In Figure 5.1 b the HC tasks are being executed on processors with better reliability such as T11 and T19 which are scheduled on more suitable processors which wasn't possible due to LC tasks being executed in the low criticality mode.

The high criticality mode schedule also goes beyond the execution time of the low criticality mode. This is dealt by dropping the LC tasks that cause the system to work beyond deadlines as the impact of LC tasks doesn't affect the correctness of the result.

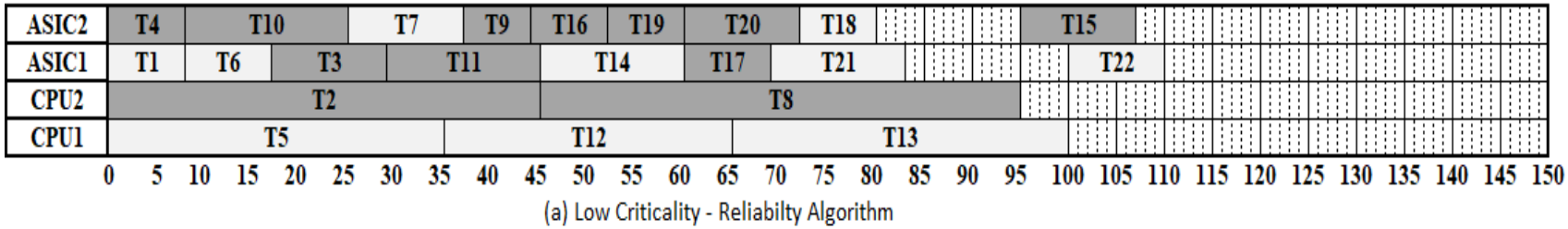


Figure 5.1 Task Schedule for Reliability Priority Apporach

CHAPTER VI:

RELIABILITY AWARE EXECUTION LATENCY REDUCTION

Reduction in latency of the system is the primary focus of the algorithms in this approach. Algorithm 3 operates in the LC mode of the system and prioritizes latency in scheduling the tasks to the available processors. Whereas algorithm 4 operates in HC mode processing HC tasks ahead of LC tasks thereby providing access to faster processors. Reduction of HC task latency is prioritized over LC task Execution

6.1 Low Criticality System Mode

Following a similar procedure from the previous approach the LC mode of the system all the tasks are grouped into a set. The tasks with the greatest number of successors to the sink are singled out. If there are multiple tasks fitting the criteria, they are further filtered out with other characteristics such as criticality, distance from source and latency on the available processors. This process is repeated until we remain with a single task and assign it to the best available processor for the task depending on the approach used.

Algorithm 3 takes the input data. In steps 1 to 4, it compiles the data and arranges tasks in the order of their predecessors. From steps 5 to 18 the algorithm selects the task that has the highest priority and assigns it to the fastest processor, the rest of the algorithm loops back to select the next task to be scheduled

Algorithm 3: Scheduling with Low Criticality System M ode

INPUT: Task graph with 'n' tasks, Tech Library with (R_{ji}, L_{ji}) where j ∈ J (1,4) refers to processors and i ∈ I (1, n) denotes the tasks in task graph,
OUTPUT: Schedule of LC reliability priority mode, L_o, R_o, L_{hc}, R_{hc}

1. Update $I_{current}$ = {i | i are tasks that are available to be scheduled, i ∈ I}
2. Update $J_{current}$ = {j | j are processors that are idle, j ∈ J}
3. Form the set of 'I′ where
4. I = {i | i are the tasks with the greatest number of successors, i ∈ $I_{current}$}
5. **if** |I| = 1
6. **then** $T_{current} = T_i$
7. **else** I′ = I
8. I = {i | i is High Critical task, i ∈ I′}
9. **if** |I| = 1
10. **then** $T_{current} = T_i$
11. **else** I′ = I
12. I = {i | i is nearest to the source, i ∈ I′}
13. **if** |I| = 1
14. **then** $T_{current} = T_i$
15. **else** I′ = I
16. I = {i | i has lowest $L_j(i)$, i ∈ I′ & j ∈ $J_{current}$}
17. **if** |I| = 1
18. **then** $T_{current} = T_i$
19. Assign $T_{current}$ to j ∋ $L_j(i)$ is the lowest, j ∈ $J_{current}$
20. **if** | $J_{current}$ | = 0
21. **then** wait for time T
22. Update $I_{current}$ & $J_{current}$
23. **if** | $I_{current}$ | = 0
24. **then** wait for time T
25. Update $I_{current}$ & $J_{current}$
26. Repeat step 4 until | I | = 0 i.e. all the tasks are scheduled

6.2 High Criticality System Mode

In Algorithm 4, the first four steps separate the tasks as HC and LC. Then, steps 5 to 15 select the highest critical task and assigns it to the fastest processor. Steps 16 to 22 loops back until all the HC tasks are scheduled. The steps form 23 till the end schedules the rest of the tasks accordingly.

Algorithm 4: Scheduling with High Criticality System Mode

INPUT: Task graph with 'n' number of tasks, Tech Library with (R_{ji}, L_{ji}) where j ∈ J (1,4) refers to processors and I ∈ I (1, n) denotes the tasks in task graph,

OUTPUT: Schedule of Low Criticality reliability priority mode, L_o, R_o, L_{hc}, R_{hc}

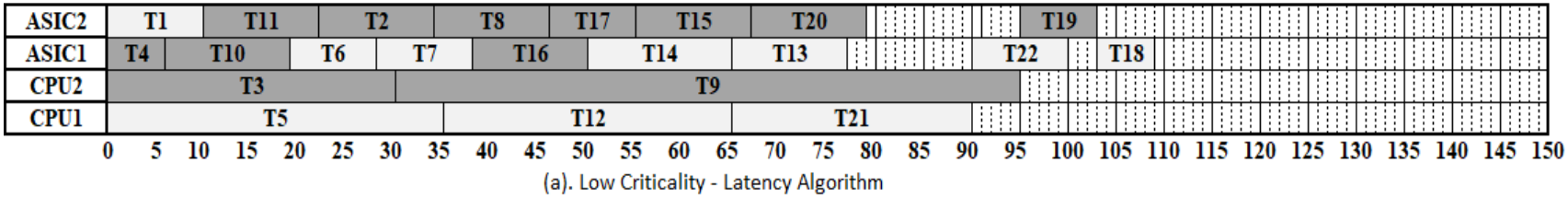
1. Update $H_{current}$ = {h | h are High Critical tasks that are available to be scheduled, h ∈ I}
2. Update $J_{current}$ = {j | j are processors that are idle, j ∈ J}
3. Form a set of 'H' where
4. H = {h | h are the tasks with the greatest number of successors, h ∈ $H_{current}$}
5. **if** |H| = 1
6. **then** $T_{current} = T_h$
7. **else** H′ = H
8. H = {h | h is nearest to the source, h ∈ H′}
9. **if** |H| = 1
10. **then** $T_{current} = T_h$
11. **else** H′ = H
12. H = {h | h has lowest $L_j(h)$, h ∈ H′ & j ∈ $J_{current}$}
13. **if** |H| = 1
14. **then** $T_{current} = T_h$
15. Assign $T_{current}$ to j ∋ $L_j(h)$ is the lowest, j ∈ $J_{current}$
16. **if** | $J_{current}$ | = 0
17. **then** wait for time T
18. Update $H_{current}$ & $J_{current}$
19. **if** | $H_{current}$ | = 0
20. **then** wait for time T
21. Update $H_{current}$ & $J_{current}$
22. Repeat step 3 until | H | = 0 i.e. all the High Critical tasks are scheduled
23. Form a set of tasks $L_{current}$ where
24. $L_{current}$ = {l | l are the remaining tasks that are available to be scheduled with greatest number of successors, l ∈ I}
25. **if** |L| = 1
26. **then** $T_{current} = T_l$
27. **else** L′ = L
28. L = {l | l is nearest to the source, l ∈ L′}
29. **if** |L| = 1
30. **then** $T_{current} = T_l$
31. **else** L′ = L
32. L = {l | l has lowest $L_j(l)$, l ∈ L′ & j ∈ $J_{current}$}
33. **if** |L| = 1
34. **then** $T_{current} = T_l$
35. Assign $T_{current}$ to j ∋ $L_j(l)$ is the lowest, j ∈ $J_{current}$
36. **if** | $J_{current}$ | = 0

37. **then** wait for time T
38. Update $L_{current}$ & $J_{current}$
39. **if** | $L_{current}$ | = 0
40. **then** wait for time T
41. Update $L_{current}$ & $J_{current}$
42. Repeat step 23 until | L | = 0 i.e. all the tasks are scheduled

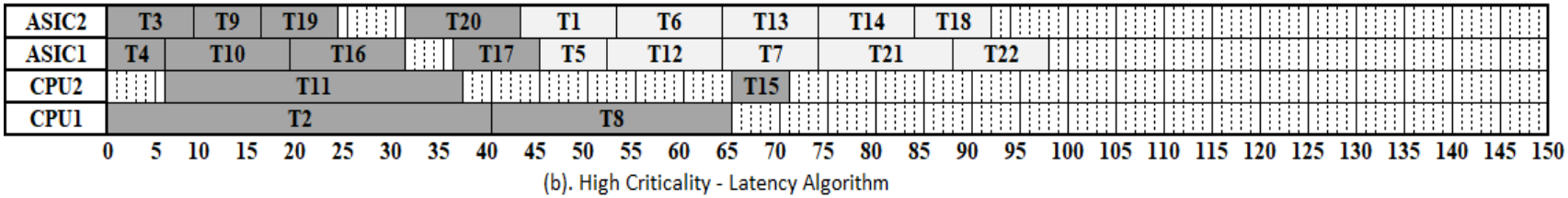
We repeat the above steps until all the tasks are scheduled. When tried with the motivational example the resulting schedule is presented in Fig 6.1.a, 6.1.b.

The low criticality mode schedule executes all the tasks without considering the task criticality. This results in longer wait times for HC tasks such as T9 and longer execution times such as T9. Although this is still better latency than the reliability priority approach in Figure 5.1.a, the overall system latency still needs improvement.

In the high criticality mode of the system the drawbacks are taken care of. HC tasks are scheduled far ahead with better latency values and leave enough slack for all the LC tasks to be executed at a faster pace too. Tasks T9 and T19 are scheduled into faster processors and all the LC tasks also have access to the faster ASIC's once HC tasks are executed. This improves the HC task latency and as an added bonus also helps the overall latency of the system as well. The algorithm also watches the reliability values for the system in the priority conditions so as to not just achieve a faster execution time but also succeed with as less affect to reliability as possible.



(a). Low Criticality - Latency Algorithm



(b). High Criticality - Latency Algorithm

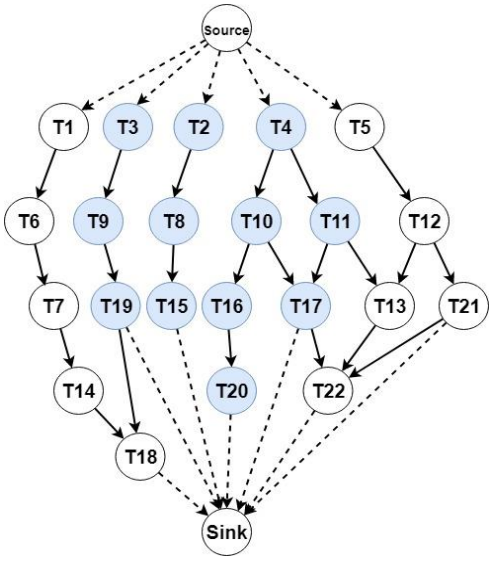
Figure 6.1 Task Schedule for Latency Priority Approach

CHAPTER VII:

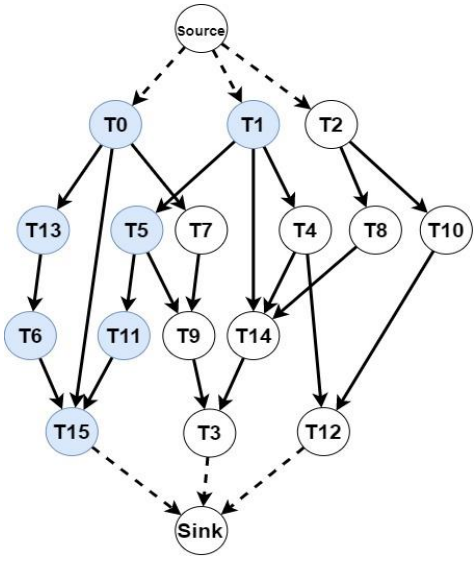
EXPERIMENTAL EVALUATION

Assuming the target architecture described in Chapter III, the proposed algorithms are tested using task graphs randomly generated using the TGFF tool. Each graph has a unique technology library containing the . The task graphs named as TG1 (Fig7.1. (a)), TG2 (Fig7.1. (b)), TG3(Fig7.1. (c)), TG4 (Fig7.1. (d)), and TG5 (Fig7.1. (e)) are depicted in the following pages.

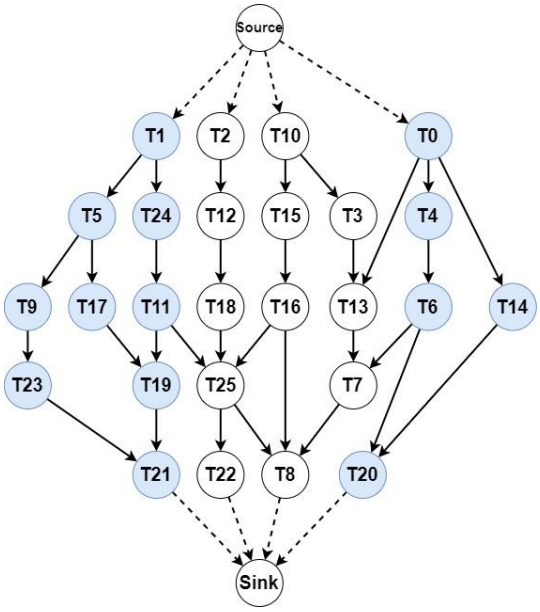
A detailed experimental evaluation using these five graphs is presented in Table 6.1. The reliability values and the execution latencies for each graph in low criticality and high criticality modes are presented along with the percentage of reliability improvement. As seen in the table, the proposed reliability priority algorithm is successful in increasing the reliability of the HC tasks by prioritizing them over other tasks. This resulted in lesser holdup times for HC tasks and making more reliable processors to be available when they are scheduled. We compare the HC task reliabilities in both system modes for each task graph to obtain reliability improvement. The algorithm aims at increasing the HC tasks' reliability to ensure the system's success in critical times and emergencies. Scheduling LC tasks after HC tasks gives HC tasks better access to reliable processors, thereby inducing an improvement in HC task reliability. The improvement is consistent in tasks with a higher number of task graphs than in graphs with fewer tasks.



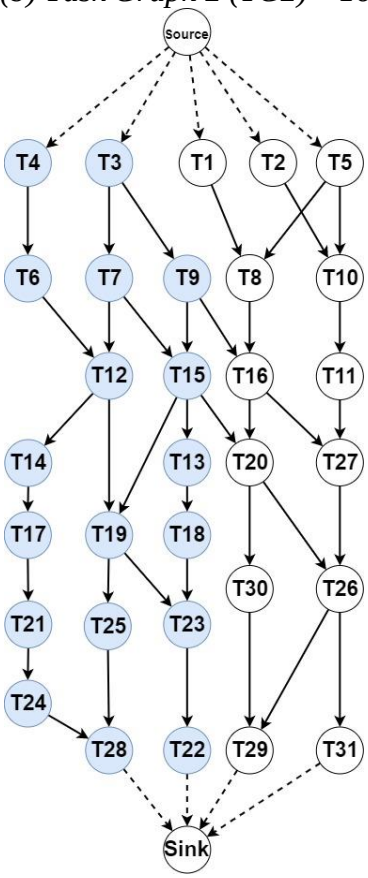
(a) Task Graph 1 (TG1) – 22 tasks



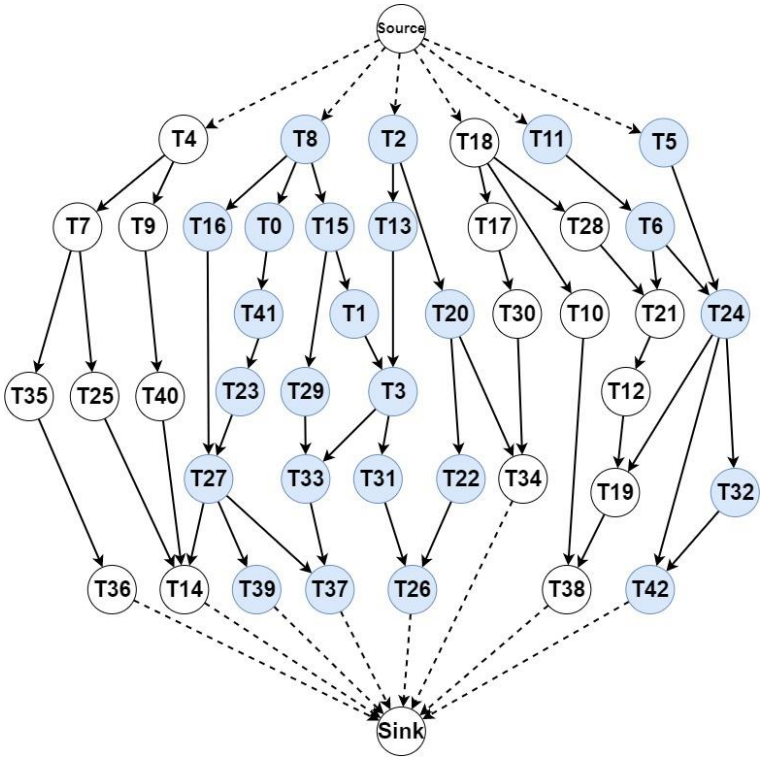
(b) Task Graph 2 (TG2) – 16 tasks



(c) Task Graph 3 (TG3) – 26 tasks



(d) Task Graph 4 (TG4) – 31 tasks



(e) Task Graph 5 (TG5) – 43 tasks

Figure 7.1 Experimental Task Graphs

TG1 is our sample task graph. TG2, which has 16 tasks, has high criticality reliability improvement of just 0.19 %. This is due to the lesser number of tasks available to schedule at any given point in the execution. Owing to this most of the tasks have the more reliable processors readily available in both operating modes. Although the algorithm returns high reliability it does not improve the reliability of HC tasks much. It is also observed that in the cases where the system reliability is increased, it comes at the cost of increased latency. HC tasks being assigned to highly reliable but slower processor cause this. However, prioritizing HC tasks in high criticality system mode ensures that all tasks are scheduled ahead of LC tasks and no stalling occurs due to a LC task occupying a better processor. This overall latency overhead can be avoided as the LC tasks can be

dropped to run within the time bound. This does not affect the primary goal of the algorithm which is improving the reliability of HC tasks in the system. The algorithms also keep track of HC task latency as achieving reliability at a huge cost to latency is not a viable option.

The overall system reliability gradually decreases as the number of tasks increases as we discussed in reliability modeling. As explained in previous sections this occurs due to the reliabilities of each task multiplying with each other thereby scaling up the error percentage along with it.

Table 6.1
Reliability Priority Approach

Task graph	Number of tasks	Low Criticality mode				High Criticality mode				% Reliability Improvement.	
		Reliability		Latency		Reliability		Latency		Overall system	High Critical tasks
		Overall	HC task	Overall	HC task	Overall	HC task	Overall	HC task		
TG1	22	0.7863	0.8548	110	107	0.8317	0.8978	162	73	6.41	4.71
TG2	16	0.7839	0.9263	155	155	0.7879	0.9281	157	157	0.51	0.19
TG3	25	0.5680	0.7301	164	162	0.6440	0.8067	255	125	13.38	10.49
TG4	31	0.6459	0.7789	243	173	0.6933	0.8933	233	171	7.34	14.69
TG5	43	0.6097	0.7549	205	205	0.6518	0.7773	245	160	6.91	2.97

The latency priority approach from the adjacent table shows a much better improvement rate of latency by mapping tasks to the fastest available processor, the system succeeds in attaining better execution latency values. This method improves latency values but watches the drop in reliability in the process. The focus is on improving the latency of HC tasks ahead of the overall system. As HC tasks are assigned to faster processors, LC tasks run on slower processors. LC tasks which decrease the latency will be dropped so that the system can proceed into the next cycle. These cases can be observed in TG3 and TG5 in the table above. In these examples, the system succeeded in decreasing execution times for HC tasks but went over the specified time. This is rectified by dropping low criticality tasks and bringing the system under the required time deadline. The results indicate that the algorithm can be utilized for systems to transfer the data collected at a faster rate when time is of utmost priority than accuracy.

Table 6.2
Latency Priority Approach

Task graph	Number of tasks	Low Criticality mode				High Criticality mode				% Latency improvement	
		Reliability		Latency		Reliability		Latency		Overall system	High Critical tasks
		Overall	HC task	Overall	HC task	Overall	HC task	Overall	HC task		
TG1	22	0.7431	0.8378	107	79	0.7769	0.8870	97	71	9.35	10.13
TG2	16	0.6963	0.8503	103	72	0.7114	0.8546	103	60	0	16.67
TG3	25	0.5314	0.7183	110	110	0.4742	0.6644	160	68	-45.45	38.18
TG4	31	0.5366	0.7122	205	200	0.5536	0.7355	178	116	13.17	42
TG5	43	0.5742	0.7045	157	135	0.5595	0.7216	200	118	-27.38	12

CHAPTER VIII:

CONCLUSION

In this thesis, we worked on improving the reliability and execution latency of high critical tasks in mixed criticality systems in the context of HW/SW codesign. The system can operate in two modes: low criticality operation mode and high criticality operation mode. We introduced two new algorithms improving each optimization metric. The proposed algorithms operating in high criticality mode outperform their low criticality mode algorithms through providing HC tasks with better choice of processors. In the reliability priority approach, the execution latency provided by the first algorithm is used as an upper bound for the second algorithm which aims at improving the reliability of high critical tasks in the system without incurring any latency overhead. The experimental evaluation conducted using automatically generated task graphs shows an average 6.61% reliability improvement for HC tasks. The latency priority approach utilizes the execution time of the system as the deadline when operating in high criticality operation mode. The proposed algorithm schedules HC tasks before looking into LC tasks and achieves a significant latency improvement (23.8% on the average) for HC tasks without exceeding the low criticality mode latency. The algorithms work better for task graphs with higher number of task graphs. Whenever a high criticality mode schedule goes beyond the latency or reliability constraints, the system rectifies by dropping LC tasks which do not affect the final output. The planned future work includes focusing on more control over choosing the tasks to deem critical and investigating the scenarios where the criticality of tasks change in the course of execution when the system is in high criticality mode.

REFERENCES

1. S. Baruah, “The Federated Scheduling of Systems of Mixed-Criticality Sporadic DAG Tasks,” 2016 IEEE Real-Time Systems Symposium (RTSS), pp. 227–236, 2016.
2. D. Tamas-Selicean and P. Pop, “Task Mapping and Partition Allocation for Mixed-Criticality Real-Time Systems,” 2011 IEEE 17th Pacific Rim International Symposium on Dependable Computing, pp. 282–283, 2011.
3. R. Medina, E. Borde, and L. Pautet, “Scheduling Multi-periodic Mixed-Criticality DAGs on Multi-core Architectures,” 2018 IEEE Real-Time Systems Symposium (RTSS), pp. 254–264, 2018.
4. M. Bagheri and G. Jervan, “Fault-Tolerant Scheduling of Mixed-Critical Applications on Multi-processor Platforms,” 2014 12th IEEE International Conference on Embedded and Ubiquitous Computing, pp. 25–32, 2014.
5. S. Vestal, “Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance,” 28th IEEE International Real-Time Systems Symposium (RTSS 2007), pp. 239–243, 2007.
6. A. Burns and R. I. Davis, “A Survey of Research into Mixed Criticality Systems,” ACM Computing Surveys, vol. 50, no. 6, pp. 1–37, Nov. 2017.
7. E. Azari and H. Koc, “Improving performance through path-based hardware/software partitioning,” 2015 Fifth International Conference on Digital Information Processing and Communications (ICDIPC), pp. 54–59, 2015.
8. S. Tosun, N. Mansouri, E. Arvas, M. Kandemir, Y. Xie, and W.-L. Hung, “Reliability-Centric Hardware/Software Co-Design,” Sixth International Symposium on Quality of Electronic Design (ISQED05), pp. 375–380, 2005.

9. B. Nimer and H. Koc, “Improving reliability through task recomputation in heterogeneous multi-core embedded systems,” 2013 The International Conference on Technological Advances in Electrical, Electronics and Computer Engineering (TAEECE), pp. 72–77, 2013.
10. P. K. Saraswat, P. Pop, and J. Madsen, “Task Mapping and Bandwidth Reservation for Mixed Hard/Soft Fault-Tolerant Embedded Systems,” 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 89–98, 2010.
11. V. Izosimov, P. Pop, P. Eles, and Z. Peng, “Synthesis of fault-tolerant embedded systems with checkpointing and replication,” Third IEEE International Workshop on Electronic Design, Test and Applications (DELTA06), pp. 5, pp. 447, 2006.
12. S. K. Baruah et al., ‘‘Scheduling real-time mixed-criticality jobs,’’ IEEE Trans. Comput., vol. 61, no. 8, pp. 1140–1152, Aug. 2012
13. C. Bolchini and A. Miele, “Reliability-Driven System-Level Synthesis for Mixed-Critical Embedded Systems,” IEEE Transactions on Computers, vol. 62, no. 12, pp. 2489–2502, Dec. 2013.
14. V. Izosimov, P. Pop, P. Eles, and Z. Peng, “Scheduling of Fault-Tolerant Embedded Systems with Soft and Hard Timing Constraints,” 2008 Design, Automation and Test in Europe, pp. 915–920, 2008.
15. M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos, “Mixed-Criticality Real-Time Scheduling for Multicore Systems,” 2010 10th IEEE International Conference on Computer and Information Technology, pp. 1864–1871, 2010.

16. H. Li and S. Baruah, “Outstanding Paper Award: Global Mixed-Criticality Scheduling on Multiprocessors,” 2012 24th Euromicro Conference on Real-Time Systems, pp. 166–175, 2012.
17. D. D. Niz, K. Lakshmanan, and R. Rajkumar, “On the Scheduling of Mixed-Criticality Real-Time Task Sets,” 2009 30th IEEE Real-Time Systems Symposium, pp. 291–300, 2009.
18. P. Penil, H. Posadas, J. Medina, and E. Villar, “UML-based single-source approach for evaluation and optimization of mixed-critical embedded systems,” 2015 Conference on Design of Circuits and Integrated Systems (DCIS), pp. 1–6, 2015.
19. A. Namazi, S. Safari, and S. Mohammadi, “CMV: Clustered Majority Voting Reliability-Aware Task Scheduling for Multicore Real-Time Systems,” IEEE Transactions on Reliability, vol. 68, no. 1, pp. 187–200, 2019.
20. R. I. Davis and A. Burns, “Priority Assignment for Global Fixed Priority Pre-Emptive Scheduling in Multiprocessor Real-Time Systems,” 2009 30th IEEE Real-Time Systems Symposium, pp. 398–409, 2009.
21. J. Theis and G. Fohler. Mixed criticality scheduling in time-triggered legacy systems. Proc. WMC, RTSS, 2013.
22. R.M. Pathan. Improving the Schedulability and Quality of Service for Federated Scheduling of Parallel Mixed-Criticality Tasks on Multiprocessors. In Sebastian Altmeyer, editor, 30th Euromicro Conference on Real-Time Systems (ECRTS 2018), volume 106 of Leibniz International Proc. in Informatics (LIPIcs), pages 12:1–12:22. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.
23. M. Hassan, “Heterogeneous MPSoCs for Mixed-Criticality Systems: Challenges and Opportunities,” IEEE Design & Test, vol. 35, no. 4, pp. 47–55, 2018.

24. Kalavade and E. Lee, “A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem,” Third International Workshop on Hardware/Software Codesign, Grenoble, France, pp. 42–48, 1994.
25. Giovanni De Micheli, Synthesis and Optimization of Digital Circuits, McGraw-Hill Higher Education, 1994
26. Theis, Jens and Gerhard Fohler. “Schedule Table Generation for Time-Triggered Mixed Criticality Systems.” (2013).
27. Alireza Namazi, Meisam Abdollahi, Saeed Safari, and Siamak Mohammadi. 2017. A Majority-Based Reliability-Aware Task Mapping in High-Performance Homogenous NoC Architectures. ACM Trans. Embed. Comput. Syst. 17, 1, Article 28 (December 2017).
28. T. Zhang, Q. Yue, X. Zhao, G. Liu, "An improved firework algorithm for hardware/software partitioning", Appl. Intell., vol. 48, no. 12, pp. 100-116, 2018.
29. C. Yao, L. Qiao, L. Zheng, and X. Huagang. Efficient schedulability analysis for mixed criticality systems under deadline-based scheduling. Chinese Journal of Aeronautics, 2014. 16
30. J. Wang and H. Wang. Work-in-progress: Scheduling of graph-based end-to-end tasks for distributed multi-criticality systems. In Proc. IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pages 129–132, 2017. 40
31. R. Trub, G. Giannopoulou, A. Tretter, and L. Thiele. Implementation of partitioned ¨ mixed-criticality scheduling on a multi-core platform. ACM Trans. Embed. Comput. Syst., 16(5s):122:1–122:21, 2017. 25

32. A. Thekkilakattil, R. Dobrin, and S. Punnekkat. Mixed criticality scheduling in fault-tolerant distributed real-time systems. In Embedded Systems (ICES), 2014 International Conference on, pages 92–97. IEEE, 2014.

33. L. Sigrist, G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele. Mapping mixed criticality applications on multi-core architectures. In Proc. DATE, pages 1–6, 2014.

34. L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode change protocols for priority driven preemptive scheduling. Journal of Real-Time Systems, 1(3):244–264, 1989.

35. R. Schneider, D. Goswami, A. Masrur, M. Becker, and S. Chakraborty. Multi-layered scheduling of mixed-criticality cyber-physical systems. Journal of Systems Architecture, 59(10, Part D):1215 – 1230, 2013.

36. Y. Zhou, S. Samii, P. Eles, and Z. Peng. Partitioned and overhead-aware scheduling of mixed criticality real-time systems. In Proc. of 24th Asia and South Pacific Design Automation Conference, ASPDAC, pages 39–44. ACM, 2019.

37. L. Zeng, P. Huang and L. Thiele, "Towards the design of fault-tolerant mixed-criticality systems on multicores," 2016 International Conference on Compilers, Architectures, and Synthesis of Embedded Systems (CASES), Pittsburgh, PA, 2016, pp. 1-10.

38. R. M. Pathan, "Schedulability Analysis of Mixed-Criticality Systems on Multiprocessors," 2012 24th Euromicro Conference on Real-Time Systems, Pisa, 2012, pp. 309-320.

39. D. Müller and A. Masrur, "The schedulability region of two-level mixed-criticality systems based on EDF-VD," 2014 Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, 2014, pp. 1-6.

40. S. Maurer and R. Kirner. Cross-criticality interfaces for cyber-physical systems. In Proc. 1st IEEE Int'l Conference on Event-based Control, Communication, and Signal Processing, 2015.

41. J. Lin, A.M.K. Cheng, D. Steel, and M.Y.-C. Wu. Scheduling mixed-criticality real-time tasks with fault tolerance. In L. Cucu-Grosjean and R. Davis, editors, Proc. 2nd Workshop on Mixed Criticality Systems (WMC), RTSS, pages 39–44, 2014.

42. Z. Li and S. He. Fixed-priority scheduling for two-phase mixed-criticality systems. ACM Trans. Embed. Comput. Syst., 17(2):35:1–35:20, 2018.

43. V. Legout, M. Jan, and L. Pautet. Mixed-criticality multiprocessor real-time systems: Energy consumption vs deadline misses. In L. George and G. Lipari, editors, Proc. ReTiMiCS, RTCSA, pages 1–6, 2013.

44. J. Lee, H.S. Chwa, L.T.X. Phan, I. Shin, and I. Lee. MC-ADAPT: Adaptive task dropping in mixed-criticality scheduling. ACM Trans. Embed. Comput. Syst., 16:163:1–163:21, 2017.

45. K. Lakshmanan, D. de Niz, R. Rajkumar, and G. Moreno. Resource allocation in distributed mixed-criticality cyber-physical systems. In ICDCS, pages 169–178, 2010.

46. P. Huang, H. Yang, and L. Thiele. On the scheduling of fault-tolerant mixed-criticality systems. In Proc. Design Automation Conference (DAC), pages 1–6. IEEE, 2014.

47. N. Guan, P. Ekberg, M. Stigge, and W. Yi. Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems. In IEEE RTSS, pages 13–23, 2011.

48. D. Socci, P. Poplavko, S. Bensalem, and M. Bozga. Time-triggered mixed-critical scheduler on single- and multi-processor platforms. In Proc. HPCC/CSS/ICESS, pages 684–687, 2015.

49. D. Socci, P. Poplavko, S. Bensalem, and M. Bozga. Mixed critical earliest deadline first. In Proc. Euromicro Conference on Real-Time Systems (ECRTS), 2013.

50. J. Ren and L.T.X. Phan. Mixed-criticality scheduling on multiprocessors using task grouping. In Proc. 27th ECRTS, pages 25–36. IEEE, 2015.