Safety-Aware, Timing-Predictable and Resource-Efficient Scheduling for Autonomous Systems

Jie Zou

Doctor of Philosophy

University of York

Computer Science

March 2023

Abstract

Advanced driver-assistance (ADAS) and semi-autonomous systems represent the major computing demands for road vehicles, which are complex and safety-critical with strict real-time and resource constraints. With an awareness of the criticality requirements of the functions, this thesis focuses on designing shared resource scheduling methods to ensure that systems comply with criticality-related timing constraints and which improve their resilience through highly efficient resource utilisation. First, a novel graceful degradation strategy is proposed for mixed-criticality contexts based on understanding task dependency from a functional perspective to handle uncertainties (e.g., timing faults) raised by conflicts for shared resources. Incorporating multiple operational modes and causality analysis-based graceful degradation, it effectively manages uncertainties and conflicts, outperforming existing methods and maximizing system-wide functional Quality of Service (QoS). Second, a novel consistent mixed-criticality multi-core task static scheduling method is developed to replace the multi-system modes multi-schedules method, which can lead to unnecessary task discarding and pose challenges during system criticality mode changes. The proposed approach introduces criticality-informed temporal isolation, enables and simplifies task-level mode changes and significantly enhances the system's resilience and the survivability of tasks, resulting in remarkable improvements in overall system performance and outperforms state-of-the-art approaches. Finally, the work is extended from the task-level to the network-level mixed-criticality data transmission. The proposed in-vehicle network scheduling method substantially improves system resilience by tolerating timing faults of safety-critical traffic. In addition, the introduced server-based method enhances bandwidth utilisation efficiency and reduces resource waste, further contributing to the overall system improvement.In summary, this thesis contributes to the resilient and efficient scheduling of shared resources in mixed-criticality systems at both the task and network levels.

Contents

Abstract **3**

List of Figures **10**

List of Tables **11**

Abbreviations **15**

Acknowledgements **17**

Declaration **20**

1 Introduction **21**

1.1 The Complexity of Autonomous Systems 23

1.2 The Complexity of Timing Requirements 25

1.3 Research Hypothesis and Objectives 29

1.4 Thesis Organisation 31

2 Background and Literature Review **33**

2.1 Complexity of Autonomous Systems 34

2.2 Timing Predictability 39

2.3 Real-Time Scheduling 41

2.4 Mixed Critically Systems 49

2.5 Single Processor Analysis for MCS 50

2.6 Multicore Scheduling and Analysis for MCS 55

2.7 Survivability and Graceful Degradation 58

2.8 Network Bandwidth Scheduling 60

2.9 Summary 65

3 **Context- and Causality-aware Graceful Degradation for Mixed-Criticality Scheduling** **67**
3.1 Introduction 68
3.2 Method Overview 71
3.3 Formulation of Importance Ordering 81
3.4 Formulation of Graceful Degradation 87
3.5 Evaluation 93
3.6 Summary 103

4 **Resilience-aware Multi-core Mixed-criticality Consistent DAG Scheduling** **105**
4.1 Introduction 106
4.2 Method Overview 108
4.3 Consistent Schedule Formulation 121
4.4 Evaluation 137
4.5 Summary 149

5 **Resilient and Efficient Time-Sensitive Network** **151**
5.1 Introduction 152
5.2 Method Overview 155
5.3 reTSN Formulation 164
5.4 Evaluation 172
5.5 Summary 186

6 **Conclusions and Future Work** **187**
6.1 Contributions 189
6.2 Limitations and Constraints 191
6.3 Future Work 193

Bibliography **197**

List of Figures

1.1 The sensing system of an autonomous car [105]. 21
1.2 The hardware architecture for an autonomous vehicle based on zone architectures and advanced processing system [60]. 24
1.3 Processing graph of an autonomous driving system with task frequencies [66]. 26

2.1 Information flow in an autonomous vehicle [110]. 34
2.2 The overview of Cartographer LiDAR Simultaneous Localisation and Mapping SLAM algorithm [68]. 35
2.3 The Vehicle System Domain Architecture [60]. 37
2.4 Typical multi-core platform. 37
2.5 Breakdown Utilisation [47] 48
2.6 The released jobs of task τ_k in the busy period of length R_i^s. . . 54
2.7 Example of streams in a typical autonomous driving system [83]. 61
2.8 The simplified architecture of in-vehicle network [113]. 62
2.9 Overview of a TSN switch [114]. 64

3.1 AAIP mobile robot end-to-end pipeline. 73
3.2 The example of a system represented by a Bayesian network. . 83
3.3 The conditional probability table of the system. 85
3.4 The updated tables of τ_1, τ_4 and τ_5. 86
3.5 The example of τ_4 sum-product based elimination. 87
3.6 The EU value and remaining *LO* tasks at each task dropping point of a specific graph. (with Util = 0.7, 3 applications, 18 tasks, of which 6 are *LO* criticality tasks) 95
3.7 The proportion of systems with higher EU value under Util = 0.6 (10 randomized graph structures and 30 CPD tests for each bar) 97

3.8 The distribution of EU value difference of improved system . . 98
3.9 The proportion of systems with improved survivability under Util = 0.6 (10 randomized graph structures and 30 CPD tests for each bar) 98
3.10 The distribution of survived percentage difference of improved system 99
3.11 The proportion of systems with higher EU value under Util = 0.9 (10 randomized graph structures and 30 CPD tests for each bar) 99
3.12 The distribution of EU value difference of improved system under Util = 0.9 100
3.13 The proportion of systems with improved survivability under Util = 0.9 (10 randomized graph structures and 30 CPD tests for each bar) 100
3.14 The distribution of survived percentage difference of improved system under Util = 0.9 101
3.15 Experiment results for the proportion of systems with higher EU value under varied Util from 0.3 to 0.9 (each bar consists of 300 trials) 101
3.16 Experiment results for the distribution of EU value difference of improved system under varied Util from 0.3 to 0.9 102
3.17 Experiment results for the proportion of systems with improved survivability under varied Util from 0.3 to 0.9 102
3.18 Experiment results for the distribution of survived percentage difference of improved system under varied Util from 0.3 to 0.9 103

4.1 The proposed system-level scheduling architecture 110
4.2 Simplified example of a dual-criticality system 111
4.3 Example of LO-criticality task segmentation in a dual-criticality system 113
4.4 Example of HI-mode schedule with degraded LO-criticality task in dual-criticality system 113
4.5 Static schedules example for multi-criticality system [93] 116
4.6 An illustrative example of a mixed schedule 118
4.7 Selected task scheduling flowchart 119
4.8 Mixed-criticality DAGs example from a dual-criticality system 121
4.9 Initialisation and update of waiting queue 121

4.10 Task allocation on mixed schedule 125
4.11 Example of task scheduling calculation in a triple-criticality system 129
4.12 The consistent schedule example for a dual-criticality system . 136
4.13 The consistent schedule example when all HI tasks overrun . . 136
4.14 The structure of object detection function 138
4.15 The schedulability of systems against normalised utilisation (g2, g3, and g4 represent systems comprised of 2, 3, and 4 DAG applications, respectively.) 142
4.16 The normalised preemption rate of systems against normalised utilisation 143
4.17 The normalised migration rate of systems against normalized utilisation 144
4.18 The survival proportion of LO tasks in systems with Uti=0.3 . 145
4.19 The discarded proportion of LO tasks in systems with Uti=0.3 145
4.20 The survival proportion of LO tasks in systems with Uti=0.6 . 146
4.21 The discarded proportion of LO tasks in systems with Uti=0.6 146
4.22 The survival proportion of LO tasks in systems with Uti=0.9 . 147
4.23 The discarded proportion of LO tasks in systems with Uti=0.9 147

5.1 TSN-based in-vehicle network architecture 153
5.2 Temporal fault caused by a delayed traffic frame 155
5.3 Frame transmission using a credit-based shaper [25] 156
5.4 The conventional AVB TSN switch 158
5.5 The constant bandwidth server based TSN switch 160
5.6 Example of multi Class A frames transmission ($T_{ser} = 23$, $C_{ser} = 8$, $iS_a = 0.375$, $sS_a = -0.625$) 163
5.7 Example of Delayed ST frame transmission 166
5.8 The number of delayed frames successfully handled by the proposed and conventional methods 174
5.9 The finish time of event-triggered frames from Class A and Class B with $U_p = 0.3$ (index starts from 1) 176
5.10 The finish time of event-triggered frames from Class A and Class B with $U_p = 0.5$ (index starts from 1) 177
5.11 The finish time of event-triggered frames from Class A and Class B with $U_p = 0.6$ (index starts from 1) 179

5.12 The finish time of event-triggered frames from Class A and Class B with $U_p = 0.8$ (index starts from 1) 180
5.13 The finish time difference of event-triggered frames from Class A and Class B with network load $U_p = 0.3$ 182
5.14 The finish time difference of event-triggered frames from Class A and Class B with network load $U_p = 0.5$ 183
5.15 The finish time difference of event-triggered frames from Class A and Class B with network load $U_p = 0.6$ 184
5.16 The finish time difference of event-triggered frames from Class A and Class B with network load $U_p = 0.8$ 185

List of Tables

4.1 Time consumed by YOLOv5s and points nodes in different scenarios. 138
4.2 Time consumed by YOLOv5s node restoration from different initial states. 140
4.3 The performance of proposed mccs-dag scheduling strategy compared with the lsai-edf method. 148

5.1 The finish time variation of event-triggered frames without delayed ST frames 172
5.2 The variation in the finish time of delayed ST frames 173
5.3 The finish time variation of event-triggered frames with delayed ST frames 181

List of Abbreviations

AI Artificial Intelligence

ML Machine Learning

ECU Electronic Control Units

AV Autonomous Vehicles

AD Autonomous Driving

ADAS Advanced Driver Assistance Systems

OTA Over-The-Air

E/E Electric/Electronic

RA Risk Assessment

SILs Safety Integrity Levels

ASILs Automotive Safety Integrity Levels

EDF Earliest Deadline First

G-EDF Global Earliest Deadline First

EDF-VD Earliest Deadline First with Virtual Deadlines

FPS Fixed-Priority Scheduling

QoS Quality-of-Service

RTA Response Time Analysis

RTS Real-Time Systems

MCS Mixed Critically System

WCET Worst-Case Execution Time

DAG Directed Acyclic Graph

SLAM Simultaneous Localisation And Mapping

vSLAM Visual-based Simultaneous Localisation And Mapping

VIO Visual Inertial Odometry

MIT Minimum Inter-arrival Time

ETS Execution-Time Server

TBS Total Bandwidth Server

CBS Constant Bandwidth Server

OPA Optimal Priority Assignment

FPPS Fixed-Priority Preemptive Scheduling

DMPO Deadline-Monotonic Priority Ordering

OPA-MLD Optimal Priority Assignment Minimising Lexicographical Distance

SMC Static Mixed Criticality

AMC Adaptive Mixed Criticality

ST Safety-critical Traffic

AVB Audio-Video-Bridging

CAN Controller Area Network

TSN Time-Sensitive Network

IVN In-Vehicle Network

PCP Priority Code Point

TAS Time-Aware Shaper

GCL Gate Control List

GS Gate State

FIFO First-In-First-Out

SMT Satisfiability Modulo Theorie

BBN Bayesian Belief Network

CPT Conditional Probability Table

EU Expected Utility

NoC Network-on-Chip

SPM Scratchpad Memory

SRAM Static Random Access Memory

ROS Robot Operating System

Acknowledgements

Foremost, I would like to profoundly express my deepest gratitude to my supervisors, Prof. John McDermid and Dr. Xiaotian Dai, for their unwavering support, expertise, and patience throughout my PhD study. Their invaluable guidance, constructive feedback, and enthusiasm for my research have been instrumental in shaping this thesis and inspiring me to find the field of study to which I want to devote my whole life. I feel incredibly fortunate to have had the opportunity to learn from and work with such knowledgeable and dedicated supervisors. Without their encouragement and understanding, this accomplishment would not have been possible.

I would also like to express my appreciation to my internal assessors, Dr. Rob Alexander and Prof. Iain Bate, for their insightful comments and suggestions that greatly enhanced the quality of my work. I would like to thank my friends and colleagues who have supported me in various ways during this journey, including but not limited to Dr. Richard Hawkins, Dr. Victoria Hodge, Matt Osborne, Gricel Vazquez, Dr. Xinwei Fang, Mrs. Chrysle Hudson and James Hidler.

To my amazing husband, Dr Hao Sun. Your understanding, patience, and encouragement have been essential. Thank you for always believing in me, even when I doubted myself, and for being my sounding board, my cheerleader, and my best friend.

Finally, my heartfelt gratitude goes out to my parents, Ying Huang and Yuanping Zou, for their unconditional love, support, and patience throughout this challenging process. Your unwavering encouragement and companionship have been the solid foundation upon which I have built my academic and personal pursuits.

Declaration

I declare that this thesis is a presentation of original work, and I am the sole author. Certain parts of the material presented within this thesis have appeared in published and submitted papers, journals and reports, viz:

- Jie Zou, Xiaotian Dai, John A. McDermid, "Resilience-aware Mixed-criticality DAG Scheduling on Multi-cores for Autonomous Systems", in International Conference on Reliable Software Technologies. 2022.
- Jie Zou, Xiaotian Dai, John A. McDermid, "reTSN: Resilient and Efficient Time-Sensitive Network for Automotive In-Vehicle Communication", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 2022.
- Jie Zou, Xiaotian Dai, John A. McDermid, "Graceful Degradation with Condition- and Inference-awareness for Mixed-Criticality Scheduling in Autonomous Systems", The 2nd Real-time and Intelligent Edge Computing Workshop. 2023.
- Jie Zou, Xiaotian Dai, John A. McDermid, "Causality- and Context-aware Graceful Degradation for Mixed-Criticality Scheduling in Autonomous Systems", submitted to IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems *(R2)*.
- Jie Zou, Xiaotian Dai, John A. McDermid, "Resilience-aware Consistent Mixed-Criticality DAG Scheduling on Multi-cores for Autonomous Systems", submitted to ACM Transactions on Cyber-Physical Systems *(R1)*.
- Jie Zou, "AAIP Robot Object Detection Safety Case and Tasks Execution Control", AAIP-internal Technical Report, Department of Computer Science, University of York, 2022.

This work has not previously been presented for an award at this or any other university. All sources are acknowledged as references.

Chapter 1

Introduction

In the last decade, the development of sensors, computational hardware platforms, and advanced algorithms such as artificial intelligence (AI) and machine learning (ML) has significantly increased the automation level of systems. Specifically, these advancements have enabled automated systems to operate in complex and dynamic environments while ensuring safety and efficiency with minimal human intervention. Figure 1.1 illustrates modern autonomous systems that connect various sensors installed in different regions. Despite these developments, there have been several reports of fatal accidents involving autonomous vehicles. The report [34] summarised the challenges of assuring the safety of highly automated systems operating in complex environments. Evidently, we are still facing significant challenges in autonomous systems before they can provide real safety benefits to society.

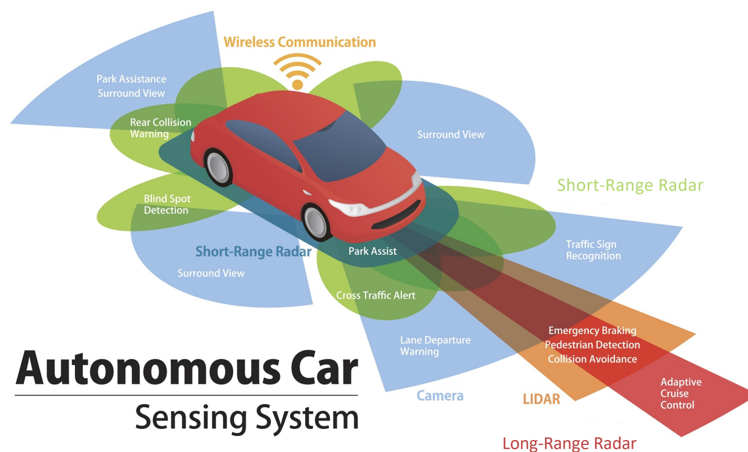

Figure 1.1: The sensing system of an autonomous car [105].

Functional safety processes and procedures are used to both ensure and assure that a system can be designed and implemented to perform safely, minimising the risk of harm to people, the environment, and property [90]. However, solely adhering to functional requirements is inadequate to ensure the safety of a system in terms of "correctness". Functional requirements primarily address questions such as "What should the system do?" and emphasise the system's behavior and capabilities. These requirements are crucial for system design, development, and testing, they are typically specific, measurable, and verifiable, but they are not sufficient to ensure safety. One of the main challenges in designing autonomous systems is addressing the non-functional requirements, which define how well a system must perform in terms of quality attributes, such as reliability, real-time performance, scalability, and security, to ensure safe and effective operation [62]. We take object detection as an example. Suppose the non-functional requirement of real-time performance is not met, caused by a delay in object detection, the functional requirement of collision avoidance could be impacted, resulting in a life-threatening accident. On the other hand, a failure in object detection can lead the system to perform extensive computation due to fault handling, which may cause a delay in the system's response and can impact the system's real-time performance, thus affecting its ability to meet the non-functional requirements. Thus, it is clear that it is crucial to simultaneously consider functional and non-functional requirements when designing and developing safety-critical systems.

With the exponentially increasing number of functions and system complexity level, research in functional domains has received considerable attention and made notable progress. Meanwhile, with the limitation of hardware resources, the increasing number of computationally intensive tasks and data with higher bandwidth requirements, we are also facing challenges in efficiently allocating system resources such as CPU time, memory, and power to ensure that the system operates reliably and meets its non-functional requirements. As the motivating example illustrates, autonomous systems' functions need temporal correctness. As P. Graydon et al emphasise in [63], the safety requirements for the timing of systems, which is directly related to real-time performance from non-functional requirements, and is a critical concern in the Real-Time Systems (RTS) research community as RTS have stringent requirements for operations to take place within specific timing constraints [33]. However, there is a gap between functional and non-functional domains. In

recent years, there has been an increasing trend of incorporating functional-related factors into work in the non-functional domain. However, these efforts often only consider simplistic uncertainties as metrics for scheduling design. For example, determining the execution order of various object detection algorithms to minimize latency and false positive rates [4]. Simple uncertainties may not adequately capture the full range of factors contributing to system performance, reliability, and safety. Therefore, it is vital to propose novel task scheduling strategies to handle the uncertainties raised by computational platforms with an awareness of the wider range of functional requirements.

It can be seen that scheduling is a crucial research consideration that includes scheduling algorithms, resource-sharing protocols and analytical methods [35]. This thesis is situated in the non-functional domain and focuses on efficiently scheduling system resources to improve the system's resilience against timing faults and Quality of Service (QoS) while maintaining safety and timeliness. The shared resources considered in this thesis are processing resources for task execution and bandwidth resources for data transmission because missing deadlines can be caused by tasks not completing their execution in a timely manner or the processing node not being able to receive the required data on time. Therefore, scheduling strategies need to be designed separately for task execution and traffic transmission to provide a comprehensive solution to tackle timing uncertainties.

1.1 The Complexity of Autonomous Systems

In a vehicle, the software runs on embedded hardware platforms known as Electronic Control Units (ECUs). Integrating new functions such as autonomous driving (AD) and advanced driver assistance systems (ADAS) has led to many vehicles having more than 100 Electronic Control Units (ECUs) with increased weight and cost [5]. As a result, the traditional domain architecture is expected to be replaced by a new generation of zone architectures that integrate different domains even further, which offers benefits such as new forms of vehicle control that were difficult to implement with independent domains, vehicle diagnosis based on combined information, and efficient Over-The-Air (OTA) software updates.

Figure 1.2 illustrates the architecture, which includes integrated ECUs running software for multiple domains and zone ECUs that gather information

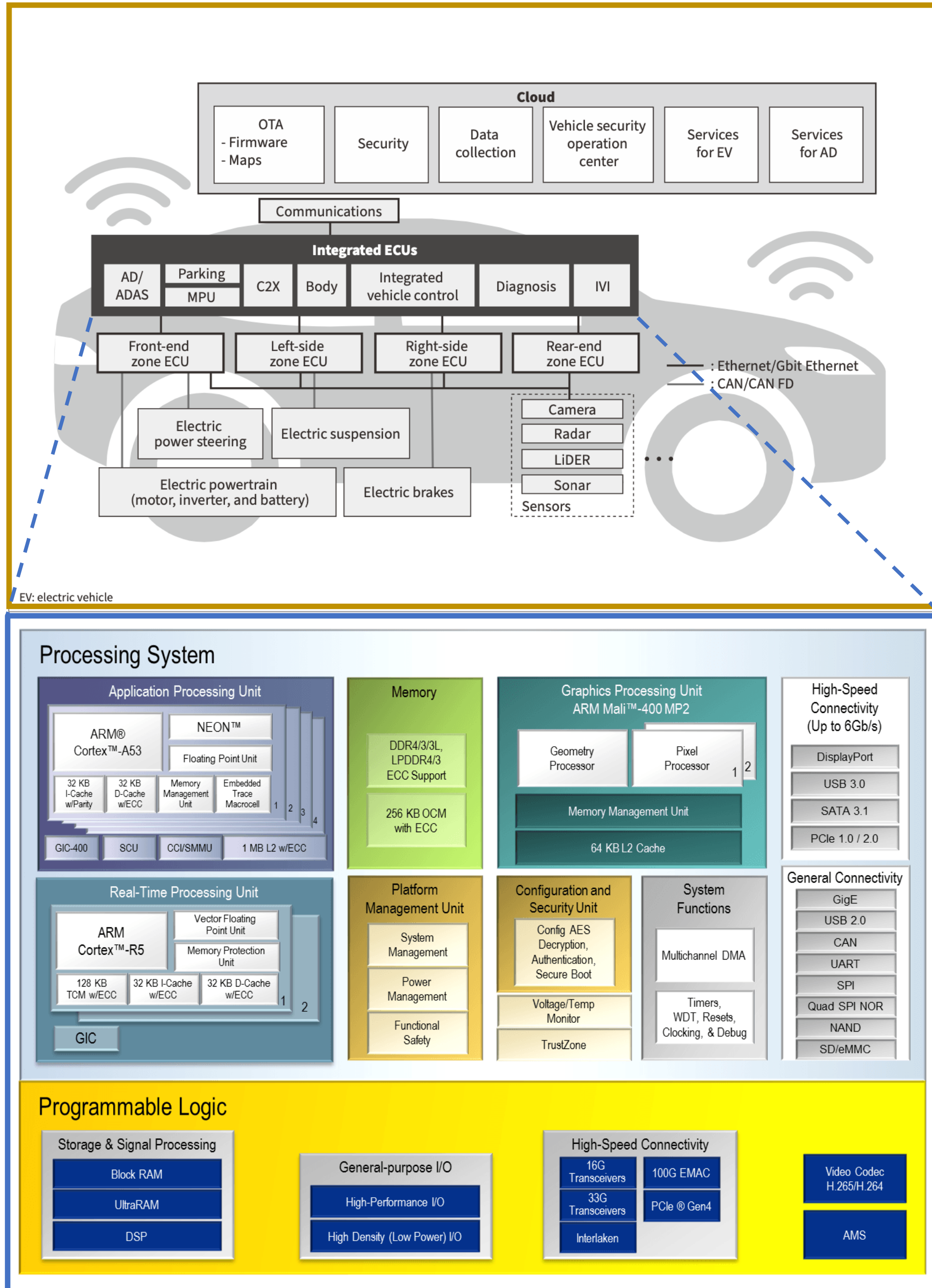

Figure 1.2: The hardware architecture for an autonomous vehicle based on zone architectures and advanced processing system [60].

from different regions of the vehicle, regardless of the domain (e.g., front or rear, left or right). This cross-domain integration of ECUs is anticipated to reduce the cost of the electric/electronic (E/E) architecture. However, this brings new challenges.

- To support the execution of advanced algorithms, various state-of-the-art computing platforms, including conventional multi-core CPU systems, GPUs, FPGAs, and ASICs, are integrated into ECUs, which need to simultaneously host multiple functions with varying functional and non-functional safety requirements.

- Similarly, since the information collected by zone ECUs combines all domains, a higher-bandwidth Ethernet backbone network with better timing control than previous architectures is necessary.

Addressing these challenges requires careful consideration of system architecture, resource allocation, fault detection and handling mechanisms, and adaptability to changing conditions. Resource scheduling strategies which recognise these considerations can improve system resilience. However, limited resources still pose challenges, especially for complex systems with functional dependencies and criticality requirements. Further, software in autonomous systems must meet timing-related safety requirements in any operational condition.

1.2 The Complexity of Timing Requirements

From the real-time perspective , autonomous systems are safety-critical with strict real-time and resource constraints. The deep processing pipeline of an autonomous driving system has end-to-end time constraints. Figure 1.3 shows a simplified example of a processing graph for an autonomous driving system with frequency information. In practice, more components can be included based on the backbone shown in the figure. The system consumes raw sensing data from mmWave radars, LiDARs, cameras, and GNSS/IMUs, with each sensor producing raw data at a different frequency. The processing components are invoked at different frequencies, performing computations using the latest input data and periodically producing outputs to downstream components [66]. The computation components (tasks) are deployed on integrated ECUs. The problem to be solved is to develop scheduling strategies and analysis

techniques to ensure that the system satisfies all timing constraints. The final control command can be executed correctly, and in a timely manner, when each component complies with timing constraints.

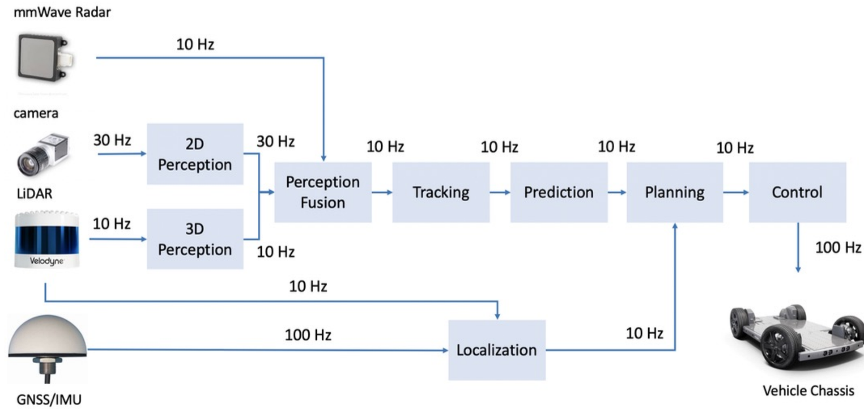

Figure 1.3: Processing graph of an autonomous driving system with task frequencies [66].

1.2.1 Complexity Introduced by Criticality Awareness

As systems with critical safety implications, their failure could have unacceptable consequences such as loss of life or significant financial damage [109]. It is impossible to prove the safety of such complex systems entirely. However, adhering to safety standards such as ISO 26262 [2] can assist in reducing the risk and mitigating the effects of system failure. These standards introduce Safety Integrity Levels (SILs) such as the Automotive Safety Integrity Levels (ASILs) A-D for ISO 26262. Identifying (A)SILs are defined as a result of conducting hazard and risk assessments. However, higher SILs can lead to substantial increases in development costs and more pessimistic system assumptions. Usually, safety-critical systems with multiple components require verification at the highest safety assurance level, which can lead to high costs and resource under-utilisation [31, 55]. For example, the worst-case execution time (WCET) of a task is a critical system parameter and is estimated based on the criticality level of the task. For the same code, if the task is defined as safety-critical, it requires a higher level of assurance, resulting in a higher WCET than it would if it is only considered to be mission-critical or non-critical [31]. This property significantly modifies/undermines many standard

scheduling results. Because of requirements relating to cost, space, weight, heat generation and power consumption, a growing trend in real-time and embedded systems is integrating components with different levels of criticality within the same system, referred to as Mixed-Criticality Systems (MCSs) [31]. In this context, the fundamental research question underlying these initiatives and standards is: how to reconcile the conflicting requirements of partitioning for (safety) assurance and sharing for efficient resource usage.

The motivation for incorporating criticality into resource scheduling is to improve the safety and reliability of autonomous systems. It is undeniable that although pessimistic estimates are inevitably introduced, in the design of scheduling strategies, considering criticality levels can prioritise allocating resources to ensure that critical tasks obtain sufficient resources to meet their requirements and deadlines. However, the actual execution behavior at run-time may not follow the worst-case scenario, and this could lead to severe resource waste because the time allocated to tasks is defined based on their criticality-dependent worst-case estimation. Therefore, we must find methods to reduce waste and utilise resources effectively to provide more flexibility to the system. This, in turn, gives rise to challenges in scheduling which are the focus of this thesis.

1.2.2 Complexity Introduced by Resilience Requirements

Resilience is essential for resource scheduling in autonomous systems. Dynamic and unpredictable environments can introduce uncertainties, such as system component failures with various causes, including hardware faults, software bugs, or external interference. Well-designed scheduling strategies should enable the system to handle uncertainties quickly, recover from unexpected events or failures, and continue operating safely. ISO 26262 includes time-related requirements, explicitly recommending monitoring execution time to detect overruns. However, it makes no connection between the quality of timing analysis and how overruns are handled [63].

Ensuring that the system can continue to operate safely and efficiently in the face of timing faults is extremely important, and that requires resilience. A resilient schedule can be achieved by allocating resources to different tasks of the system based on their criticality level so that, in the event of a failure, the more critical tasks are given priority over the less critical ones. For example, consider a complex system comprising multiple CPUs responsible for

executing various functions. Each CPU is assigned a set of tasks, with varying criticality levels. In the event of a CPU failure, a backup solution can employ a resource allocation strategy to prioritize the more critical tasks over the less critical ones. A resilient schedule may involve adjusting scheduling priorities or task assignments. This allows for the dynamic reallocation of the remaining available CPU resources to critical tasks, even if those resources were initially responsible for executing non-critical tasks. By redistributing the workload, the system can continue to operate safely even in a degraded mode, where noncritical tasks are temporarily disabled until the fault can be resolved. However, traditional system mode change drops tasks indiscriminately, leading to resource waste and a significant reduction in quality of service (QoS). The wasted resources should be allocated to tasks that can maintain the system's QoS. Thus, the research problem can be viewed as improving the system's resilience while maximising the survivability of each task.

1.2.3 Complexity Introduced by Multiple Operational Modes

Driven by the need for resilience, there is a need to design a system that can be run in different criticality modes (e.g., safety-critical, mission-critical, and non-critical). In each mode, the tasks that must be guaranteed can be defined in terms of assigned criticality, and the scheduling policies should ensure that tasks meet their timing requirements. This implies that operational mode transitions would change the schedule. It is necessary to guarantee safety during mode change. However, lack of flexibility increases the difficulty of preserving safety and "cost" of the system mode switch may increase. Switching between modes often necessitates the allocation or deallocation of specific resources, such as memory, processing power, or network bandwidth. This reallocation of resources can introduce overhead and potentially impact the overall efficiency and utilisation of the system. These challenges motivate us to find a more flexible scheduling strategy to ease mode changes whilst ensuring safely and maximising QoS.

Additionally, the operational environment can be highly dynamic and unpredictable, affecting resource availability and QoS. That means that the system can exhibit multiple operational modes depending on the environment or operational task. By incorporating environment awareness into resource scheduling strategies, the system can adjust the allocation of resources to respond to changes in the environment and ensure that critical tasks can finish

their execution on time. In that way, the system's adaptability and resilence can be improved. For example, for an ADAS, the system can adaptively allocate computing resources to different tasks based on the traffic conditions and the complexity of the road scene. When driving in heavy traffic or a complex urban environment, the system may need to allocate more computing resources to perception and planning tasks to ensure the vehicle can navigate safely and avoid collisions. On the other hand, in less challenging driving conditions, the system can allocate more resources to other tasks to improve the QoS. For instance, it can optimise the planned trajectory to improve ride comfort and fuel economy.

1.2.4 Summary

To summarise, as a mixed-criticality, safety-critical system, the resource scheduling for autonomous systems must be designed carefully to satisfy stringent timing requirements. Resource usage should be able to guarantee the execution of safety-critical tasks. Meanwhile, the system should remain resilient to ensure that autonomous systems can continue operating safely and effectively, despite disruptions, environmental changes or failures. These challenges motivate our research to design time-predictable and resource-efficient adaptive scheduling for autonomous systems to satisfy non-functional timing and safety requirements and to improve the resilience of systems.

1.3 Research Hypothesis and Objectives

The research is focused on designing a comprehensive resource scheduling strategy motivated by the non-functional safety requirements of autonomous systems, especially timing requirements, from the real-time performance perspective. To guarantee the temporal correctness of functions, the resources, including the processing unit and network bandwidth resource must be managed. Dealing with processing units is necessary to guarantee that tasks can finish their execution within deadlines, and handling network resources is necessary to guarantee that data can be received so that related processing can start on time.
Against this background, the hypothesis of this PhD is:

"A suitable resources scheduling strategy can dramatically improve the resilience

of safety-critical systems and achieve higher resource utilisation and system utility, even in the face of timing faults and operational mode changes."

To demonstrate improvement in resilience, the evaluation needs to show that, compared with traditional methods, the designed scheduling methods can tolerate timing faults raised from task execution and data transmission with higher efficiency even in a high workload situation. The methods we develop in this thesis are sensitive to the operational mode and survivability. As we will show, they can effectively adapt the scheduling strategy to achieve higher resource utilisation and system utility while ensuring that all components comply with both functional and temporal correctness.

The hypothesis is based on understanding the challenges of designing safe, time-critical, mixed-criticality adaptive autonomous systems through the scheduling of system resources. The main contribution of this thesis is three-fold:

1. Enhance the system's capability to handle timing faults of tasks with different criticality levels while taking into account system-wide functionality and employing causality analysis concepts to minimise the impact on the system's quality of service.

2. Enhance the system's ability to provide a deterministic task schedule by considering task precedence constraints and criticality-dependent timing requirements. The elimination of system mode changes and effective task-level degradation when facing timing faults improves the system's resilience and the survivability of tasks in any system mode.

3. Strengthen the resilience and efficiency of in-vehicle networks to guarantee the transmission of safety-critical traffic and address timing faults, while also considering the efficient utilisation of remaining bandwidth resources for non-safety-critical traffic transmission.

Contributions 1 and 2 tackle mixed-criticality scheduling problems at the task level to ensure that tasks are executed complying with criticality-dependent timing requirements when facing overrun of safety-critical tasks. We then extend the research on resource scheduling to the network level (Contribution 3) to enhance the resilience of data transmission while achieving highly efficient bandwidth utilisation. In this thesis, we extensively use simulation to evaluate our methods as:

- Using synthetic task sets allows us to test algorithms under controlled conditions, where we can manipulate task parameters and workload to evaluate performance across a wide range of scenarios. This is often not feasible in a real-world environment.

- Synthetic task sets can help us to compare the performance of different scheduling algorithms. By evaluating multiple scheduling algorithms on the same synthetic task set, the strengths and weaknesses of each algorithm can be identified, and the designer can choose the best one for a particular use case.

- Using synthetic task sets to evaluate scheduling algorithms allows us to obtain reliable and repeatable results that can be statistically analysed, which is important to demonstrate the feasibility and generalisability of the proposed scheduling methods.

1.4 Thesis Organisation

This chapter has presented a general introduction to the background to, and the focus of, this thesis. The remaining contents of this thesis are organised as follows:

- **Chapter 2: Background and Literature Review.** This chapter introduces the background to the work. The complexity of autonomous systems will be discussed from both software and hardware perspectives. An overview of real-time scheduling methods for complex embedded systems is provided, including timing predictability, real-time scheduling principles, and mixed-criticality scheduling. Additionally, scheduling methods for in-vehicle networks are also introduced.

- **Chapter 3: Context- and Causality-aware Graceful Degradation for Mixed-Criticality Scheduling.** This chapter introduces a novel graceful degradation strategy in a mixed-criticality context that is both causality- and context-aware. A degradation case study will be introduced based on a robot developed by the Assuring Autonomy International Programme (AAIP). The evaluation of the proposed method is based on simulations.

- **Chapter 4: Resilience-aware Multi-core Mixed-criticality Consistent DAG Scheduling.** This chapter introduces a novel multi-core mixed-criticality consistent Directed Acyclic Graph (DAG) static scheduling method denoted as (*mccs-dag*), enabling and simplifying task-level mode change. The task execution control methods for the AAIP robot based on ROS 2 will be introduced as a motivation example. The evaluation of the proposed method is, again, based on simulations.

- **Chapter 5: Resilient and Efficient Time-Sensitive Network for Automotive In-Vehicle Communication.** This chapter extends the scheduling problem from computational to communication resources. One novel method is introduced, which can effectively use bandwidth resources to deal with delayed safety-critical traffic frames while improving the data transmission efficiency for frames with relatively low critical levels. Once more, the evaluation of the proposed method is undertaken using simulations.

- **Chapter 6: Conclusions and Future Work.** The last chapter of this thesis presents conclusions. The contributions are summarised and re-emphasised. Some possible future work, extending and refining the work presented in this thesis, are explored and discussed.

Chapter 2

Background and Literature Review

An autonomous system is a safety-critical and complex embedded platform that integrates an increasing number of functions based on the design requirement. It operates independently without direct human intervention and performs various tasks to function effectively in dynamic environments. Furthermore, the deployment of functions to hardware platforms often involves the use of Mixed-Criticality Systems (MCS), enabling different software components with varying levels of criticality to coexist while maintaining safety and reliability, which depend not only on the logical and functional correctness of the software but also on the timeliness of the output. Section 2.1 introduces the complexity of autonomous systems from both software and hardware perspectives, providing an intuitive illustration of the components that need to be considered when designing resource scheduling strategies. Section 2.2 and 2.3 then introduces works on shared resource scheduling, including timing predictability and real-time scheduling principles, followed by an overview of real-time scheduling methods for mixed-criticality systems in Section 2.4. Then, the research is extended from the task to the network level in Section 2.8. Finally, a summary will be given at the end of this chapter, and the problems with existing methods are discussed, which need to be investigated in this work.

2.1 Complexity of Autonomous Systems

In this section, the challenges in autonomous systems will be thoroughly introduced, starting with software and hardware complexity, followed by industrial challenges from a real-time system perspective. These challenges motivate the need to guarantee the temporal correctness of components through the scheduling of system resources.

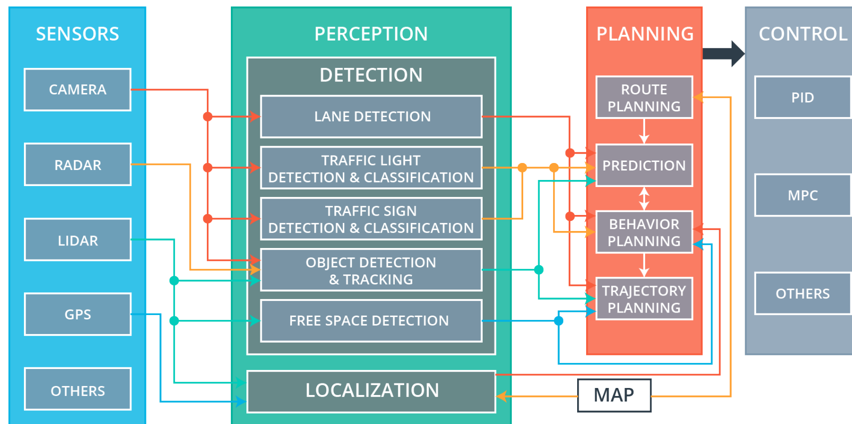

Figure 2.1: Information flow in an autonomous vehicle [110].

2.1.1 Software Complexity

As illustrated in Figure 1.1, modern autonomous systems are implemented as distributed systems, connecting various sensors installed in different regions, function nodes, and actuators. Based on the raw data produced by sensors, the autonomous software system performs the necessary processing to detect the environment and make operational decisions to guide and control the vehicle to the expected destination. As Figure 2.1 shows, except for the sensors, the software system consists of three main components: perception, planning, and control. Environment perception [73, 98, 104], high-precision localization [76, 96, 106], and fusion technologies [27] can help the vehicle understand its driving environment accurately, such as recognizing and tracking critical objects, and calculating its position at the decimeter level. Based on that, the planning model, including mission, behavior, and motion planning, can generate the target point and optimized path [102]. Finally, the tractable trajectory is fed

to the control module to operate the vehicle and follow the planned path [102].

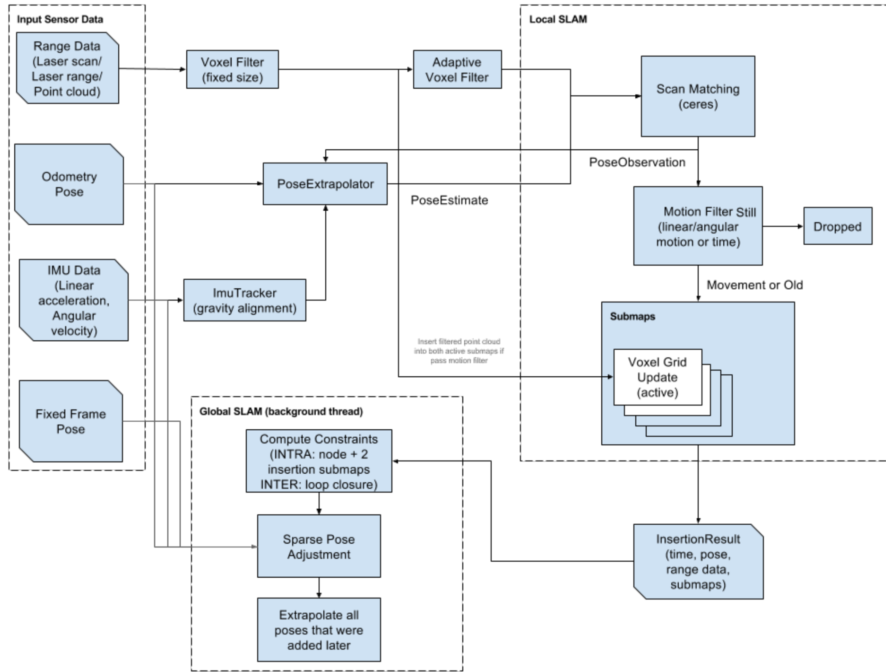

Figure 2.2: The overview of Cartographer LiDAR Simultaneous Localisation and Mapping SLAM algorithm [68].

Applications in autonomous systems have a high degree of interdependency, from perception to control. Additionally, it's worth noting that some applications, such as localization, comprise multiple functions (e.g., GPS-, Vision-, and LiDAR-based Localization), which can be further segmented into sub-functions. An example of such segmentation is the LiDAR-based simultaneous localization and mapping (SLAM) algorithm, Cartographer [68]. Figure 2.2 shows the algorithm's overview, where each box represents a processing component (sub-function). It's apparent that Cartographer consists of several sub-functions with dependencies. Moreover, the output of *PoseExtrapolator* can provide pose estimation on its own and can be fused with the result from visual-based SLAM and GPS-based methods to provide a more accurate pose estimate. Unlike other computing workloads, autonomous systems have a deep processing pipeline with strong dependencies between different stages and various localities. The term "deep processing pipeline" refers to a computational workflow or series of stages that involve multiple complex operations. These operations are typically interconnected and dependent on each other, where

the output of one stage serves as the input for the next stage. The pipeline can be visualized as a sequential flow of data or information through different stages or processing units. Regarding the term "various localities", it refers to the distributed nature of data and processing within an autonomous system. Autonomous systems often incorporate multiple sensors and actuators distributed across different physical locations. Each locality may have its own set of sensors or processing units responsible for collecting and analyzing data from their specific environment or area of interest. This thesis focuses on task scheduling, where a task is considered a minimum execution unit that cannot be further segmented into sub-functions. The following content will use the terms "task" and "function" interchangeably.

2.1.2 Hardware Complexity

As introduced in Section 1.1, the software in a vehicle typically runs within ECUs, which are embedded hardware platforms responsible for controlling different subsystems within a vehicle. Figure 2.3 illustrates a domain architecture composed of functional units called domains, which is a common practice in the automotive industry. The domains are named based on the subsystem they control, such as the powertrain, chassis, and autonomous driving/advanced driver assistance system (AD/ADAS) domains. For both the zoon architecture illustrated in Figure 1.2 and domain architecture described in Figure 2.3, a processing system (e.g., AD-ECU domain) exists to support the execution of advanced algorithms. Such ECUs comprise various state-of-the-art computing platforms, including GPUs, FPGAs, ASICs, and multi-core CPU systems. Except for sensor data e.g., high resolution images, dense point-cloud data, etc., the AD and ADAS functions also require a map position unit (MPU) to provide topographical and road information up to several kilometres ahead. This map information must be regularly updated to maintain its accuracy [60]. Therefore, in addition to the complexity of the execution platform, inter-domain communication and monitoring the network for unauthorized access make the resource management problems more complex.

Multi-Core Platforms

As one of the state-of-the-art computing platforms, multi-core CPU systems are widely used in ECUs, which is the target platform in this thesis. In a computing system, the central processing unit (CPU) and the graphics processing

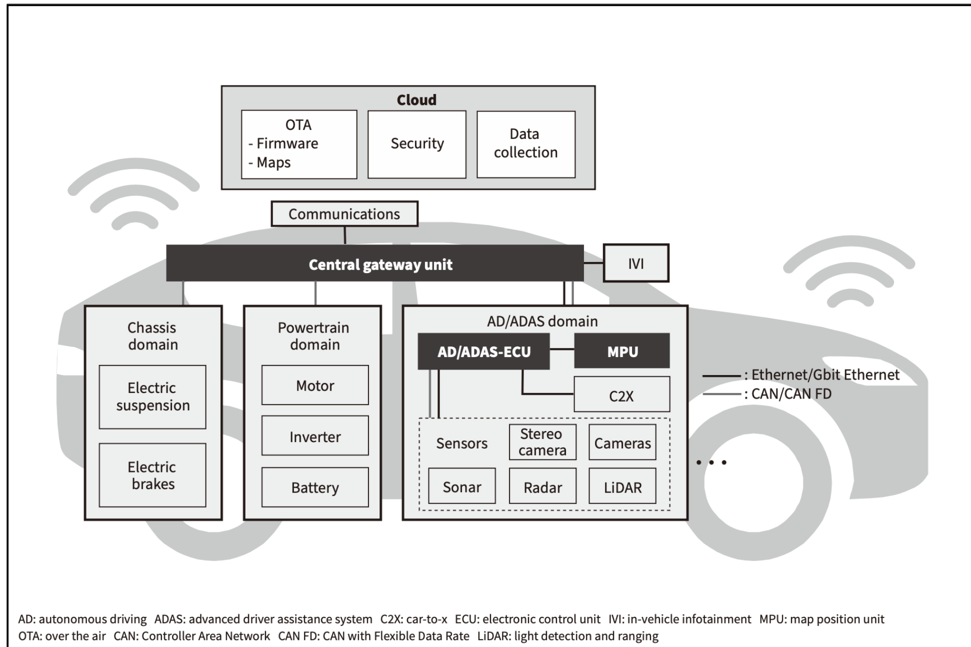

Figure 2.3: The Vehicle System Domain Architecture [60].

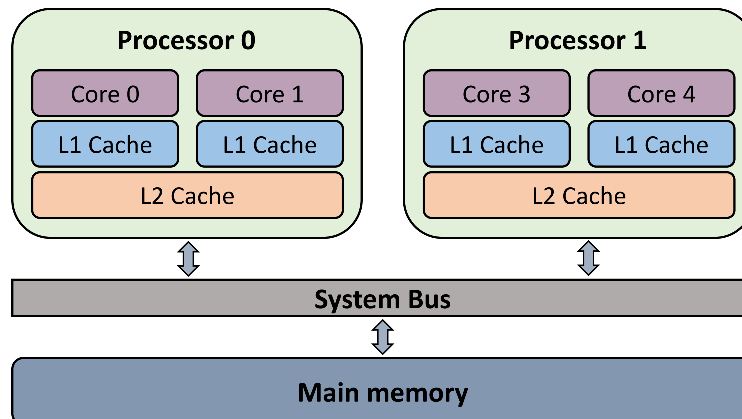

Figure 2.4: Typical multi-core platform.

unit (GPU) serve different purposes. The CPU is responsible for executing general-purpose tasks and managing system resources, while the GPU is specialized in parallel processing and handling graphics-related computations. When it comes to accessing GPUs, it's typically the CPU that coordinates and schedules tasks for the GPU to execute. The CPU acts as the control unit, managing the overall execution flow and allocating resources as needed. This coordination between the CPU and GPU is essential for leveraging the power of GPUs effectively. To limit the complexity of the system model, we assume that if a task's execution needs to be accelerated by GPUs, the estimated execution time includes the GPU accessing delay and the time consumed by GPUs.

Figure 2.4 shows a typical multi-core platform consisting of multiple processing cores that share access to a hierarchical memory system [38]. The memory hierarchy comprises multiple levels of memory components, with each level having a different capacity and access latency. The register file is at the top of the hierarchy, which is the fastest and smallest memory component. It is located within each processing core and stores the data actively used by the core. The next level of the hierarchy is the L1 cache, which stores frequently accessed data and instructions, providing quick access to the data needed by the cores. The L2 cache is a larger but slower memory component shared by all cores. It stores data and instructions that are not frequently accessed and serves as a backup for the L1 cache. Beyond the L2 cache, the platform may have an additional cache or main memory levels that have greater capacity but longer access latency. These memory components are shared by all cores and store data and instructions that are less frequently accessed than those stored in the L1 and L2 caches. As we move down the hierarchy, the capacity of the memory components increases while the access latency increases.

In summary, advanced hardware platforms can facilitate the performance of autonomous systems by guaranteeing the understanding of real-time circumstances and reacting to them fast enough. Work by Mody et al. [94] mentioned that two factors determine the reaction time of an autonomous driving system:

1. **Processing latency:** Processing latency determines how fast the system can react to the captured sensor data. In this thesis, the latency involves the instruction execution time and the memory accessing time.

   – *"How to manage processing resources to guarantee the execution of tasks that comply with timing constraints?"*

2. **Frame rate:** Frame rate determines the speed or frequency at which new data samples are collected or made available for processing. It includes the sampling time and the time for data transmission in the communication network.

   – *"How can limited bandwidth resources be effectively allocated to ensure data can be transmitted to destinations, satisfying different latency requirements?"*

2.2 Timing Predictability

In real-time systems, time predictability refers to the ability to guarantee that a task will complete within a specified time interval or deadline, which is essential for ensuring that the system can meet its timing requirements and avoid any potential hazards or malfunctions. The worst-case execution time (WCET) estimation of each task is an essential parameter for time predictability analysis because it provides an upper bound on the task's execution time (i.e., the maximum amount of time it could take to complete under any circumstances). By knowing the WCET of a task, the system can allocate enough time for its execution and ensure that it will meet its deadline. Thereby, the system can achieve time predictability and ensure that the system operates safely and reliably [64].

In other words, the principal timing-related safety evidence is timing analysis. Developers determine the Worst Case Execution Time (WCET) of each task, compute the Worst Case Response Times (WCRT) of the task set, and then use the results to show that timing requirements will be met (barring hardware failure). As L. Zhao et al introduced in paper [125], each task shows some variation in execution time depending on the input and the environment's state, which increases the difficulty of WCET estimation with sufficient confidence. This problem gets worse as processing units grow more complex. Approaches for WCET determination can be divided into three categories:
Static (analysis-based) approaches: involve analyzing the source code of a program to determine its maximum execution time under all possible input scenarios [64]. These techniques are based on analyzing the program's control flow, data flow, and memory access patterns to derive an upper bound on the execution time of each program segment. One of the most common static analysis-based techniques for WCET estimation is path analysis, which

involves identifying all possible paths through the program and determining the execution time of each path [54]. However, these techniques may produce conservative estimation, as they assume worst-case conditions for all program variables and inputs. Additionally, these techniques may not account for certain runtime behaviours, such as cache effects or hardware interrupts, which can affect program execution time.

Dynamic (measurement-based) approaches: rely on profiling the execution of a task on the target hardware to measure its actual execution time. Dynamic WCET analysis techniques can be split into three categories: high water mark, probabilistic, and search-based. High Water Mark (HWM) approaches involve executing the task multiple times while measuring its execution time and then selecting the maximum value as the WCET. However, it is not generally possible to determine the likelihood or degree of underestimation [64]. Probabilistic approaches use statistical methods to determine the WCET based on the collected data [46, 50]. However, high confidence is expensive, and the lack of questioning the validity of the data used to form the distribution makes the result uncertain [123]. Search-based approaches involve using optimization techniques to generate input test cases that will lead to the maximum execution time of the program [46, 78, 122]. Such approaches can provide accurate estimates of the WCET of a task. However, they can be time-consuming and require a large number of input test cases to be generated and executed. Dynamic WCET estimation has the advantage of providing accurate results, as it takes into account the actual execution time on the target hardware. However, it can be time-consuming and may only cover some possible execution scenarios, resulting in underestimating the true WCET.

Hybrid approaches: combine both static and dynamic methods to determine the WCET. This approach aims to leverage the advantages of both methods while minimizing their drawbacks. The key idea behind this approach is to identify the parts of the code where static analysis is more accurate and those parts where dynamic analysis can provide a more accurate estimate [124]. For example, static analysis can be used to determine the worst-case execution time of loops with a fixed number of iterations, while dynamic analysis can be used to measure the execution time of loops with a variable number of iterations. The results of the two methods can then be combined to obtain a more accurate estimate of the loop's WCET. Hybrid approaches have been

shown to be effective in reducing the overestimation of WCET estimates by static analysis and the underestimation of WCET estimates by dynamic analysis. However, the implementation of hybrid approaches can be complex and time-consuming. Additionally, the accuracy of the estimate heavily depends on the accuracy of the static and dynamic analysis tools used.

2.3 Real-Time Scheduling

As emphasized in Section 1.2, the main concern of real-time systems is meeting their timing requirements and ensuring that tasks are completed within their deadlines. This is because real-time systems are designed to respond to events and inputs within a specific timeframe, and missing a deadline can result in system failure or even dangerous consequences. To satisfy timing constraints, real-time systems must be designed, analyzed, and implemented to ensure that tasks can always meet their deadlines reliably and safely at run-time, which is generally achieved by real-time scheduling, including theories, algorithms, and methodologies for scheduling tasks and resources. This means that the system must allocate its resources, such as CPU, memory, and bandwidth, to ensure that tasks have the necessary resources to execute within their deadlines. This includes allocating sufficient resources to tasks, prioritizing tasks based on their criticality, and avoiding resource contention.

2.3.1 Task Model

As introduced in Section 2.1.1, a task has specific functionality and is invoked by an internal/external event or a timer interrupt. When extending to the communication part, a task can be regarded as data/message, and its release can be time-triggered (i.e., with a constant period) or event-triggered. Here, we introduce the task with a general approach.

A task model is a representation of the behaviour of tasks in a real-time system, which describes the tasks' characteristics. Real-time schedulers use the task model to decide which task to execute next and how much processing time and resources to allocate to each task. The task model also helps analyse the system's performance and verify its temporal correctness. A typical model for task τ_i includes the following parameters:

- **Job ($j_{i,k}$):** is a single release of a task and is the basic unit that can be scheduled by a task scheduler. $j_{i,k}$ means the kth released job of task τ_i.

- **Release time (r_i):** is the time when a task becomes available to be executed. If τ_i is a periodic task, the kth job's release time can be worked out as $r_{i,k} = (k-1)T_i + \phi(0)$, where $\phi(0)$ is the phase, i.e. the release time of the first job.

- **Worst-case execution time (WCET) (C_i):** the maximum time that a task can take to execute under any circumstance. The estimation methods have been introduced in Section 2.2.

- **Relative Deadline (D_i):** temporal constraints for tasks on completion time after their releases. Hard deadlines are those that, if missed, will jeopardize the system and may lead to disastrous consequences. Conversely, soft deadlines are those that, if missed, will only degrade the performance of the system but will not cause damage to the environment or the operator. In addition to hard and soft, there is also a firm deadline, which is defined as a deadline that can be missed, but the utility of the result will be unusable once after the deadline.

- **Period (T_i):** the time between two consecutive arrivals of the same task. In this thesis, we assume $D_i = T_i$. In simple models, there is an underlying assumption that tasks are released in a perfectly periodic manner [33].

- **Criticality (L_i):** a measure of the importance of a task, which can be used to determine the task's priority in the system.

Tasks in a real-time system can be categorised into three classes:

1. **Periodic tasks** have a fixed arrival time, execution time, and deadline, which makes them predictable and easier to schedule. The period of a periodic task is usually a constant value that is specified at design time. The deadline of a periodic task is typically equal to its period, meaning that the task must be completed before its next release.

2. **Aperiodic tasks** do not have a fixed or regular arrival time. They are typically triggered by external events or interrupts, such as user input, sensor readings, or network messages. Unpredictable arrival time makes scheduling them more challenging compared to periodic tasks.

3. **Sporadic tasks** are aperiodic in nature but have minimum inter-arrival time constraints. This means a sporadic task can occur at any time, but

there must be a certain minimum amount of time between two consecutive instances of the task. In terms of task scheduling, they always have a predefined minimum inter-arrival time (MIT).

2.3.2 Scheduling Algorithms

Scheduling algorithms are a fundamental component of real-time systems, which determine how tasks are executed and resources are allocated to meet timing constraints. According to the scheduling decision time, the schedulers can be broadly classified into static and dynamic scheduling approaches.

Static scheduling

In static scheduling, the assignment of resources to tasks is determined at system design time, and it remains unchanged during runtime [112]. This approach is typically used for systems with relatively simple requirements and limited variability in task demand. One example of static scheduling is Fixed-Priority Scheduling (FPS). In FPS, each task is assigned a fixed priority level based on its criticality, with the highest priority being assigned to the most critical task. FPS provides a simple and efficient scheduling algorithm, as the system knows the exact priority order of the tasks at design time. However, it requires an accurate estimation of the worst-case execution time (WCET) of each task to ensure that tasks meet their deadlines. In addition, if the system experiences any changes, such as adding a new task or changing the task's priority, the entire system must be re-analysed and redesigned.

Dynamic scheduling

In dynamic scheduling, the resources for task allocation is determined at runtime based on the system state and task demands. Dynamic scheduling is typically used for systems with more complex requirements and more variability in task demand. One example of dynamic scheduling is the earliest deadline first (EDF) scheduling algorithm [87], which assigns priorities based on the absolute deadline of the task. The idea behind EDF is to always schedule the task with the closest absolute deadline, which ensures that the task with the shortest time remaining to its absolute deadline is executed next. Unlike Fixed-Priority Scheduling, EDF does not require a priori knowledge of task execution times, making it more flexible and easier to implement in

systems with unpredictable task behaviour. One disadvantage of EDF is that it does not inherently provide mechanisms to address priority inversion, where a lower-priority task holds resources required by a higher-priority task. Although priority inheritance protocols can be implemented alongside EDF to mitigate priority inversion, it may introduce the risk of task starvation. In this scenario, a low-priority task may never be executed because higher-priority tasks with earlier deadlines continue to arrive. Besides, EDF has a less predictable behaviour that is known as a "domino effect" [35], i.e., a cascade of deadline misses.

For both static and dynamic scheduling, algorithms are further divided into two categories: preemptive and non-preemptive.

- **Preemptive scheduling algorithms:** In preemptive scheduling, a higher-priority task can interrupt the running task. This type of scheduling is used in systems where the highest priority task must be executed immediately. The most common preemptive scheduling algorithm is the priority-based scheduler, where the highest priority task is scheduled first.

- **Non-preemptive scheduling algorithms:** In non-preemptive scheduling, once a task is started, it cannot be interrupted until it completes. Non-preemptive scheduling is suitable for systems where it is essential to guarantee a minimum CPU time for each task. The most common non-preemptive scheduling algorithm is First-Come-First-Served (FCFS) scheduling, where the first task to arrive is the first to be scheduled.

Each method has advantages and disadvantages, and the choice of scheduling approach depends on the system's specific requirements. Dynamic scheduling approaches are generally more flexible and can handle more complex systems. In contrast, static scheduling approaches are more deterministic and can provide better guarantees on the timing behaviour of the system.

Execution-Time Server

An Execution-Time Server (ETS) is a concept used in real-time systems and scheduling algorithms. It refers to a specific type of server or task execution mechanism that guarantees timely and deterministic execution of tasks. Periodic tasks typically arise from sensory data acquisition, control loops, action

planning, and system monitoring, representing the system's primary computational demand. Four basic algorithms, Timeline Scheduling, Rate Monotonic, Earliest Deadline First, and Deadline Monotonic, are commonly used for handling periodic tasks. However, in practice, the arrival pattern of some tasks is not periodic. These tasks are often triggered by external events. Neither static nor dynamic scheduling algorithms can manage periodic and non-deterministic tasks well simultaneously. When dealing with hybrid task sets, the main objective of the kernel is to guarantee the schedulability of all critical tasks in worst-case conditions and provide good average response times for soft and non-real-time activities.

Suppose event-driven aperiodic tasks with critical timing constraints are expected to be guaranteed offline. That can be done only by making proper assumptions, that is, by assuming a maximum arrival rate for each critical event. In this thesis, aperiodic tasks characterised by a minimum inter-arrival time between consecutive instances are called sporadic tasks. The assumption can limit the sporadic load, and they are guaranteed under peak-load situations by assuming their maximum arrival rate. To handle sporadic and aperiodic tasks in a predictable way considering the Quality-of-Service (QoS) [87] requirements, a mechanism, namely Execution-Time Server (ETS), is introduced in paper [112], which is based on the principle of reserving a fixed amount of processing time for each task to guarantee its execution . In an ETS, each task is assigned a fixed time slice, which is the maximum amount of time the task can use to execute. Server-based scheduling methods can be further categorised into Fixed-Priority Servers and Dynamic Priority Servers.

One example of fixed-priority servers is the polling server; that is, a periodic task whose purpose is to service aperiodic requests as soon as possible. A server is characterised by a period T_s and a computation time C_s, called server capacity or server budget. In general, the server is scheduled with the same algorithm used for the periodic tasks, and once released, it serves the aperiodic requests within the limit of its budget. Once the budget is exhausted, the polling server is suspended and has to wait for the next release. One of the main advantages of the Polling Server for Task Execution algorithm is its simplicity and ease of implementation. However, the capacity will be wasted if no task is available when the server is released. To solve the limitation of the polling server, many solutions are proposed, e.g., Priority Exchange, Deferrable Server, and Sporadic Server [35]. These methods use principles

similar to the polling server but can reduce wasted capacity by preserving it if no tasks are available. These three algorithms are differentiated in preserving and replenishing capacity, and they use different feasibility analyses to determine the maximum capacity of the server. For example, the Sporadic Server algorithm is used to schedule tasks with unpredictable arrival and execution times. This algorithm can provide high predictability and efficiency in managing task execution for systems with unpredictable workloads. However, as a fixed-priority server, when the server has a long period, the execution of the aperiodic requests can be delayed significantly because the server is always scheduled with a far deadline. A shorter period can be allocated to a Sporadic Server to reduce the aperiodic response times. However, the capacity has to be reduced proportionally to keep the server utilisation constant. That causes more frequent replenishments and increases the number of context switches with the periodic tasks, increasing the algorithm's run-time overhead.

To overcome the drawbacks of fixed-priority servers, dynamic priority servers were designed for EDF, which can reduce the aperiodic response times by assigning an earlier deadline. The assignment must be done so that the overall processor utilisation of the aperiodic load never exceeds a specified maximum value U_s. That is the main idea behind the *Total Bandwidth Server* (TBS), a simple and efficient aperiodic service mechanism to safely schedule aperiodic requests under EDF [35]. The server's name comes from the fact that each time an aperiodic recommendation enters the system, the total bandwidth of the server is immediately assigned to it, whenever possible. One major disadvantage of the TBS is that it relies on the knowledge of the worst-case execution time specified by each job at its arrival. When such knowledge is unavailable, unreliable, or too pessimistic, hard tasks are not protected from transient overruns occurring in the soft tasks and could miss their deadlines. To avoid any hard deadline, the deadline assignment rules adopted by the server must be carefully designed. The *Constant Bandwidth Server* (CBS) can be efficiently used because it performs comparable to that of the TBS and provides temporal isolation. The basic idea behind the CBS mechanism can be explained as follows: when a new job enters the system, it is assigned a suitable scheduling deadline, which can keep its demand within the reserved bandwidth. If the job tries to execute more than expected, its deadline is postponed (i.e., its priority is decreased) to reduce interference with the other tasks. By postponing the deadline, the task remains eligible for execution.

The CBS behaves as a work-conserving algorithm, exploiting the available slack efficiently (deadline-based), thus providing better responsiveness than non-work-conserving algorithms.

2.3.3 Response-Time Analysis

Response-Time Analysis (RTA) is a formal method that provides a rigorous and systematic approach to predict the worst-case response time of tasks based on their timing characteristics and the scheduling policy used in the system. The analysis assumes that tasks are scheduled according to a fixed-priority scheduling policy, and each task's execution time is deterministic and known a prior [14]. If the worst-case response time of a task is less than or equal to its deadline, then the task is guaranteed to meet its deadline. However, if the worst-case response time of a task exceeds its deadline, then the task is said to be infeasible, and the system is deemed unschedulable.

As introduced in [33], the standard recursive equation for response time analysis is formed of two parts: (1) the worst-case execution time and (2) the interference time resulting from tasks with higher priorities.

$$\begin{aligned} R_i &= C_i + I_i \\ &= C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil * C_j \end{aligned} \tag{2.1}$$

RTA is an essential tool for designing and verifying the correctness of real-time systems. It enables system designers to ensure that the system's timing constraints are met and can be used to identify performance bottlenecks and optimize system performance.

2.3.4 Priority Assignment

For systems scheduled with fixed-priority scheduling (FPS), the task with the highest priority can be selected for execution first. As paper [47] emphasised, with poor priority assignment, the scheduler may run jobs in a far from optimal order. That will lead to severe deadline misses, even though the system workload/utilisation is still low, as shown in Figure 2.5. Suppose a system can only use a small portion of its capacity before the jobs start to miss deadlines. Then, with the requirement to add functions or an increase in jobs' execution time, the system's reliability will decrease, and there will be a need to upgrade

to more hardware, with the possibility of hardware overprovisioning. Moreover, less headroom can be utilised for additional functionality and to handle uncertainties when the hardware resources are limited. Obviously, priority assignment is one of the critical steps in scheduling systems.

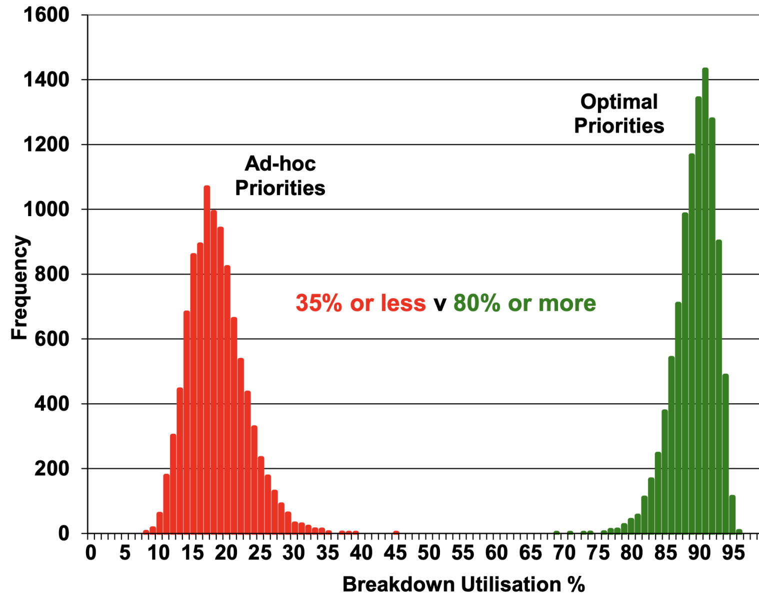

Figure 2.5: Breakdown Utilisation [47]

The conventional priority assignment method for fixed-priority preemptive scheduling (FPPS) is Deadline-Monotonic Priority Ordering (DMPO) proposed in paper [84]. DMPO is optimal for some simple systems; however, minor changes to the assumptions (for example, allowing offset release times, deadlines greater than periods, non-preemptive, or deferred preemption scheduling) break this optimality. Paper [12] proposed Audsley's Optimal Priority Assignment (OPA) Algorithm to address the drawbacks of DMPO, which can provide an optimal priority ordering. It is worth noting that Audsley 's OPA algorithm is applicable only if the schedulability test S (e.g., worst-case response time analysis introduced in Section 2.3.3) satisfies the following conditions [45]:

1. The tasks in the system are independent.

2. When the priorities of any two tasks of adjacent priority are swapped, the task assigned the higher priority cannot become unschedulable according to test S, if it was previously schedulable at the lower priority.

Typically, Audsley's OPA algorithm focuses on achieving schedulability for all of the tasks in a system under regular operation assumptions and considering each task's fairness. Optimal Priority Assignment Minimising Lexicographical Distance (OPA-MLD) algorithm proposed in paper [42] is a way of placing the most critical tasks at the highest priority levels while still maintaining schedulability.

2.4 Mixed Critically Systems

As emphasized in Section 1.2.1, the design of real-time and embedded systems, particularly those utilised in the implementation of autonomous systems, is currently witnessing a trend of integrating components with varying levels of criticality on a single hardware platform. Furthermore, these platforms are transitioning from single-core to multi-core and many-core architectures in the future. Criticality denotes the level of assurance necessary for a system component to prevent failure. Mixed criticality systems (MCS) comprise two or more levels of criticality, including safety-critical, mission-critical, and low-critical, with up to five levels possible based on standards such as DO-178B [3] and ISO 26262 [2]. At each level, tasks that require assurance can be defined according to their allocated criticality. Appropriate scheduling policies should be adopted to ensure that the tasks meet their criticality-dependent timing requirements [31]. Worst-case execution time (WCET) is one of the vital parameters, and its estimation is different under different system modes based on their safety requirements and directly influences the schedulability of the system. In this section , we introduce the most general model.

2.4.1 System Model of an MCS

As introduced in paper [120], the WCET estimate depends on the level of certification of the application or system component. The higher the criticality level, the more conservative the bound tends to be. Thus, the value of the estimated execution time is more significant, with less tolerance for deadlines being missed. Generally speaking, tasks with criticality requirements may have a set of WCETs corresponding to different confidence levels. Paper [63] summarised several WCET assessment approaches. The definition of one task becomes $(T_i, D_i, \vec{C_i}, L)$, where $\vec{C_i}$ is a vector of estimated WCETs, one per criticality level. Another critical property of MCS is that the system can

run under different criticality modes. Theoretically, the number of modes can be defined arbitrarily according to the requirements of the system designer, like the model represented in paper [51]. Take dual-criticality systems as an example; there are two system modes: low LO-criticality mode and high HI-criticality mode. The tasks executed under the HI mode are defined as HI-criticality tasks, otherwise regarded as LO-criticality. Each LO-criticality task has only one estimated WCET $C(LO)$, and each HI-criticality task has two WCET estimates: $C(LO)$ and $C(HI)$ ($C(LO) \leq C(HI)$). Then, the task can be defined by the tuples $(T_i, D_i, C_i(HI), C_i(LO), Li)$, where L_i is either LO or HI. If L_i is LO, $C_i(HI) = 0$. Note that, in a dual-criticality system, all safety-critical tasks are regarded as HI-criticality tasks, which must be guaranteed in any circumstance. Therefore, the LO-criticality tasks can be sacrificed to protect the execution of HI-criticality tasks. Initially, all tasks are executed under the LO-criticality mode. If the actual execution time of any HI-criticality task attempts to exceed $C(LO)$, then the system may change to the HI-criticality mode. The utilisation of the standard mixed-criticality system has a strict concept of critical change, such as paper [29]. After a mode change, LO-criticality tasks are suspended or dropped (active jobs are allowed to complete), and HI-criticality tasks are allowed to run to their maximum, $C(HI)$.

2.5 Single Processor Analysis for MCS

In this Section, we look at MCS schemes that are based on applying Response-Time Analysis (RTA). Regarding fixed priority scheduling, Vestal's approach proved that using Audsley's priority assignment algorithm [13] is optimal [117]

2.5.1 Static Mixed Criticality (SMC)

According to Vestal's approach, priorities of high and low-criticality tasks are able to be interleaved to provide flexibility in scheduling. Equation (2.2) shows the response time analysis equation for Vestal's approach named SMC-NO (meaning no runtime support required), where L_i represents the criticality. Although Vestal's approach does not require any runtime monitors, all tasks had to be evaluated as if they were of the highest criticality, which can be quite expensive. According to the analysis introduced in paper [19], the pessimism of Vestal's approach is mainly caused by the criticality inversion, which happens

when a *LO*-criticality task has a higher priority than a *HI*-criticality task. In this case, the *HI*-criticality mode WCET estimated values of the *LO*-criticality tasks to calculate the interferences. Such problems can be eliminated with the help of a runtime monitor. The improved response time analysis named SMC-run (meaning with runtime monitors) can be rewritten as Equation (2.3). Three cases need to be considered, depending on whether the arbitrary higher priority task τ_j has an equal, higher or lower criticality than τ_i:

1. If $L_i = L_j$, then the tasks are at the same criticality level, and the normal representative value $C_j(L_j)$ shall be used.

2. If $L_i < L_j$, then it is unnecessary to use the large value of computation time represented and $C_j(L_i)$ shall be used since the lower level of assurance is needed for task τ_i.

3. If $L_i > L_j$, then criticality inversion happens. One approach here would be to use $C_j(L_i)$, and then the algorithm works as Vestal's original model, which allows τ_j to execute for far longer than the task is assumed to do at its own criticality level. Moreover, all low-criticality tasks are required to be verified to the highest levels of importance, which would be prohibitively expensive. Instead, $C_j(L_j)$ can be used, but the runtime monitor must to used to ensure that τ_j shall not execute for more than this value.

$$R_i = C_i(L_i) + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \times C_j(L_i) \tag{2.2}$$

$$R_i = C_i(L_i) + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \times C_j(min(Li, Lj)) \tag{2.3}$$

2.5.2 Adaptive Mixed Criticality (AMC)

Adaptive Mixed Criticality (AMC) analysis is a further extension and is widely used and recognised as the most effective approach and has become the standard approach for fixed priority-based MCS. If a *HI*-criticality task executes for more than $C(LO)$ (but no greater than $C(HI)$), then, under SMC, *LO*-criticality tasks continue to execute but may miss their deadlines; but under AMC, all *LO*-criticality tasks are suspended to guarantee *HI*-criticality

tasks. In this way, some unschedulable cases in SMC will be schedulable in AMC [24]. With both analysis methods, HI-criticality tasks can continue to meet their deadlines, which implies AMC dominates SMC in terms of schedulability. For a dual-criticality system, the schedulability analysis consists of three phases: [32]:

1. In LO-criticality system mode, the schedulability of all tasks with $C(LO)$ should be verified;
2. In HI-criticality system mode, the schedulability of all tasks with $C(HI)$ should be verified;
3. All tasks with either $C(LO)$ or $C(HI)$ should be schedulable during mode change.

The original paper on AMC [24] proposes two forms of response-time analysis: AMC-rtb and AMC-max.

AMC-rtb

For the schedulability test of LO-criticality mode, the response time of analysed task τ_i can be calculated as:

$$R_i^{LO} = C_i(LO) + \sum_{j \in hp(i)} \left\lceil \frac{R_i^{LO}}{T_j} \right\rceil \times C_j(LO) \tag{2.4}$$

Where $hp(i)$ denotes the set of tasks, and their priority is higher than the analysed task. For the HI-criticality mode, only HI-criticality tasks need to be analysed:

$$R_i^{HI} = C_i(HI) + \sum_{j \in hpH(i)} \left\lceil \frac{R_i^{HI}}{T_j} \right\rceil \times C_j(HI) \tag{2.5}$$

$hpH(i)$ represents the set of HI-criticality tasks with a priority higher than or equal to the analysed task. For the analysis of mode change,

$$R_i^{*} = C_i(HI) + \sum_{\tau_j \in hpH(i)} \left\lceil \frac{R_i^{*}}{T_j} \right\rceil \times C_j(HI) + \sum_{\tau_k \in hpL(i)} \left\lceil \frac{R_i^{LO}}{T_k} \right\rceil \times C_k(LO) \tag{2.6}$$

$hpL(i)$ is the set of LO-critical tasks with a priority higher than or equal to the analysed task. If the mode change event impacts task τ_i, then the mode change time point should be smaller than R_i^{LO}. Equation 2.6 provides the response

time upper-bound of the analysed task, which includes the interference from LO-critical during mode change. However, the response time analysis could be a pessimistic estimation and reduce the schedulability of the task set. The reason is that jobs of LO-critical tasks may not execute for the entire busy period of a HI-criticality task in the LO mode. After the mode change point, there are no newly released LO jobs. Besides, for HI-critical tasks, jobs released before the mode change point only contribute no more than $C(LO)$ value. AMC-rbt-based response time analysis is computationally efficient but less accurate.

AMC-max

AMC-max can provide a more precise analysis by considering all possible mode change points and looking for the point with the maximum response time to reduce the pessimistic interference. The worst-case response time analysis for each task in the LO-criticality mode is the same as AMC-rbt. Assume that the system mode changes at time point s, which could be any point in time interval $(0, R_i^{LO}]$. R_i^s represents the response time of task τ_i when system criticality mode changes at time s. And it can be calculated according to Equation 2.7.

$$R_i^s = C_i(HI) + I_L(s) + I_H(s, R_i^s) \tag{2.7}$$

$I_L(s)$ indicates the interference from LO-criticality but high priority tasks. $I_H(s, R_i^s)$ describes the interference from HI-criticality high priority tasks within the busy period of τ_i. After time point s, LO-critical tasks will not release any new jobs. Their interference can be bounded by:

$$I_L(s) = \sum_{j \in hpL(i)} \left(\left\lfloor \frac{s}{T_j} \right\rfloor + 1 \right) \times C_j(LO) \tag{2.8}$$

For HI-criticality high-priority tasks, we need to consider the jobs released in the busy period of length R_i^s. As mentioned before, in the worst case, jobs released before time points contribute $C(LO)$; only jobs released after mode change can contribute interference of $C(HI)$.

$$M(k, s, R_i^s) = \min\left\{ \left\lceil \frac{R_i^s - s + D_k}{T_k} \right\rceil, \left\lceil \frac{R_i^s}{T_k} \right\rceil \right\} \tag{2.9}$$

$M(k, s, R_i^s)$ estimates the maximum number of jobs released by task τ_k, which contribute $C(HI)$ in the busy period of length R_i^s after mode change. Figure 2.6 displays the released jobs intuitively.

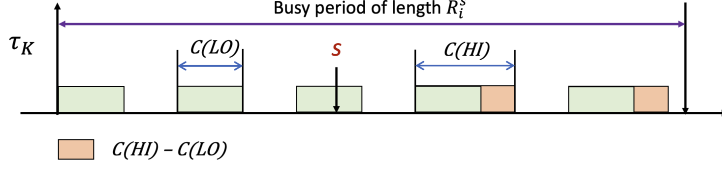

Figure 2.6: The released jobs of task τ_k in the busy period of length R_i^s.

The interference term $I_H(s, R_i^s)$ is bounded by:

$$I_H(s, R_i^s) = \sum_{k \in hpH(i)} \left\lceil \frac{R_i^s}{T_k} \right\rceil \times C_k(LO) + \sum_{k \in hpH(i)} M(k, s, R_i^s)(C_k(HI) - C_k(LO)) \tag{2.10}$$

The worst-case response time of task τ_i with mode change time point s can be given by:

$$\begin{aligned} R_i^s =& C_i(HI) + \sum_{j \in hpL(i)} (\left\lfloor \frac{s}{T_j} \right\rfloor + 1) \times C_j(LO) + \sum_{k \in hpH(i)} \left\lceil \frac{R_i^s}{T_k} \right\rceil \times C_k(LO) \\ &+ \sum_{k \in hpH(i)} M(k, s, R_i^s)(C_k(HI) - C_k(LO)) \end{aligned} \tag{2.11}$$

Each possible mode change point s in the time interval $(0, R_i^{LO}]$ should be checked to find the worst-case response time of the analysed task.

$$R_i = \max\{R_i^s\}(\forall s, s \leq R_i^{LO}) \tag{2.12}$$

Recently, the schedulability analysis for AMC-max has been extended to analyse the tasks that exhibit semi-clairvoyant behaviour [30] , which refers to a state or characteristic in which an entity or system possesses partial or limited knowledge or foresight about future events or outcomes. The notion of semi-clairvoyance was proposed in paper [6]. A fully clairvoyant scheduler, which refers to a scheduling algorithm or mechanism that possesses complete knowledge or foresight about the future behavior or events in a system, is ideal and not practicable. A semi-clairvoyant scheduler only requires information as to which mode of operation of a task will invoke. For example, some sensors can be used to perceive the change in the driving scenario and system mode change would be predicted, which implies potential timing conflicts. A semi-clairvoyant scheduler enables an earlier switch from normal to abnormal mode

and improves the system's safety. In addition to Fixed Period Scheduling (FPS), Baruah et al. [18] proposed the first scheduling algorithm focused on dynamic priority scheduling in MCS called Earliest Deadline First with Virtual Deadlines algorithm (EDF-VD). Then, based on EDF-VD, paper [40, 88] introduced the imprecise and flexible MC models, respectively. Besides, they analysed the feasibility of MCS via utilisation-based methods.

2.6 Multicore Scheduling and Analysis for MCS

J. H. Anderson et al. discussed mixed-criticality in the context of multi-core platforms in paper [8], and the extended version [95] identified Five levels of criticality from level-A (the highest) to level-E (the lowest). Tasks of criticality A are scheduled using a cyclic executive scheduling approach (a table-based method). Partitioned preemptive EDF is used to schedule Level B tasks because it has relatively low overheads and has been theoretically shown to be optimal on single-core. Levels C and D are scheduled using a Global Earliest Deadline First (G-EDF) scheduler. Finally, level E tasks are scheduled whenever the processor is idle. In the rest of this section, we introduce the multi-core mixed-criticality scheduling methods without and with the consideration of task precedence constraints.

2.6.1 Mixed-Criticality Multi-Core Scheduling

Most multiprocessor mixed-criticality scheduling algorithms can be categorized into global or partitioned methods. To maximize the utilisation of the cores, Li et al. proposed a global scheduling method for mixed-criticality systems which extends the *EDF-VD* algorithm [23] from uniprocessors to multiprocessors using a multiprocessor global scheduling method called *fpEDF* [22], which was designed for non-mixed-criticality systems. Such global scheduling algorithms allow scheduling decisions at every time quantum, leading to increased migration and preemption overheads.

Baruah et al. [20] pointed out that partitioned scheduling provides better system schedulability than global scheduling. Considering partitioning-based multiprocessor scheduling of mixed-criticality task sets, Kelly et al. [77] identified the problems of task allocation and priority assignment under fixed-priority scheduling. They investigated the schedulability of various partitioning heuristics (first-fit, worst-fit, and best-fit) and task order policies (decreas-

ing criticality and decreasing utilisation).

First-fit: First-fit works by scanning the list of available resources (e.g., memory blocks or processor cores) from the beginning and allocating the first available resource that satisfies the request.

Best-fit: Best-fit allocates the smallest available free capacity of resources to a requesting process. The algorithm searches each core for the smallest free capacity that is big enough to accommodate the task's requirement.

Worst-fit: After scanning the state on each core, the worst-Fit algorithm allocates the task to the resources with the largest free capacity. It behaves similarly to BF but goes towards the exact opposite goal.

Decreasing Utilisation: Regarding decreasing utilisation method, tasks with high utilisation values are allocated first. However, each MCS task is associated with multiple utilisation values. A single utilisation value is required to represent the task for sorting requires.

Decreasing-criticality: According to the decreasing-criticality method, tasks are ordered according to criticality and tasks at the same criticality level are further ordered by decreasing nominal utilisation.

After tasks are allocated to specific cores, priority assignment methods (e.g. Rate Monotonic and Audsleys Optimal Priority Assignment) can be used to assign the task priority on each processor for different task orders. According to the principle of the partitioned method, tasks can not be migrated to another core at run-time. This drawback may cause the system to suffer from low utilisation.

Ren et al. [108] emphasized the strong isolation among the high-criticality tasks. Each high-criticality task is associated with a subset of the low-criticality tasks and encapsulates them in a task group. The server-based strategy enhanced the isolation between tasks with different criticality and among the high-criticality tasks. In addition, the consideration of the demand for LO-criticality tasks improved the efficiency of resource allocation. However, the strategies mentioned above did not consider task dependency requirements, potentially leading to conflict between tasks with precedence constraints and severe cache misses, significantly reducing the efficiency of shared resource usage. This is especially critical for autonomous systems, where the functions have different levels of dependency, and the relationships among the tasks in each function (task group) can be complex.

2.6.2 Graph-based Mixed-Criticality Scheduling

An appropriate graph model is critical for designing a scheduling strategy which needs to consider precedence constraints. Directed Acyclic Graph (DAG) models are widely used in research on multi-core scheduling. Such models allow intuitive identification of the parallelizable part of tasks in the system, which can help to improve resource usage [127]. Medina et al. [91] proposed a novel approach based on *List Scheduling* (LS) to schedule a single DAG of MC tasks for multi-core architectures. Methods have been proposed to tackle multiple mixed-criticality DAGs using federated schedulers [17, 86, 103]. However, the task model used in [86, 103] is very restrictive. Each DAG comprises tasks from the same criticality level, i.e., HI or LO. In real complex systems, task dependencies exist between HI- and LO-criticality tasks. More practical models have been developed, i.e., LO-criticality and HI-criticality jobs may coexist in the same DAG, but HI tasks can not depend on LO tasks [17]. Once this assumption is broken, the schedulable rate, which refers to the rate or frequency at which tasks or processes can be successfully scheduled and executed within a given system or scheduling algorithm, would decrease dramatically. Medina et al. [92] adopted the DAG model of [17] and proposed a global scheduling-based method to calculate fixed schedules for HI-mode and LO-mode separately, which outperforms the original [17] in terms of the number of cores required to schedule a system and gives better schedulability when the number of cores is fixed. However, the DAG model they used also has the same restriction, i.e., high-criticality tasks can not depend on lower-criticality tasks, which is too restricted. Low-to-high communications often take place in safety-critical systems [93]. Later they started to relax this restriction and proved the generality of their method, which was the first work on scheduling multi-core MC DAG systems with multiple criticality levels. However, the migration and preemption costs are very high.

In summary, most existing graph-based multi-core scheduling methods for MCS adopt static scheduling, which provides different schedules under different system criticality modes. Once a mode change happens, a different schedule implies inevitable task migration costs because tasks would be executed on different cores in different modes. Moreover, the static schedule calculation is typically based on the classic mixed-criticality assumption, i.e., once mode change happens, all tasks with lower criticality values are dropped or suspended. The dropped task re-allocation would require more effort, and

the problem becomes more complex when considering precedence constraints. Furthermore, system recovery would not be easy, and we need to wait until a system idle time slot and unnecessarily postpone the restoration of dropped tasks. However, lower-criticality tasks could be mission-critical and play essential roles in improving the QoS of the system. Although methods have been proposed to realise task-level mode change and provide run-time strategies to minimise LO-critical task loss [28, 82], they assumed no dependency between LO and HI tasks. Further, the proposed method is only suitable for the dual-criticality system.

2.7 Survivability and Graceful Degradation

As Burns and Baruah emphasised in [29], run-time survivability of LO-criticality tasks is one of the essential problems for MCS. Later work distinguished two forms of survivability – robustness and resilience [32]. Robustness emphasises providing full functionality even when temporal faults occur. On the other hand, resilience refers to the extent to which an acceptable QoS can be achieved in the presence of timing faults. From a non-functional perspective, the most critical components (i.e., HI-criticality tasks) must continue to meet their timing requirements.

In [65], a system-wide total low-criticality budget is calculated offline for all the high-criticality applications. This budget can be allocated to the high-criticality applications according to the proposed run-time strategy based on their actual execution requirement to postpone a system mode-switch (LO to HI) as long as possible. This approach can provide a minimum budget for LO criticality tasks. However, all LO criticality tasks are treated equally in that work, and the approach only satisfies the timing requirements (it doesn't consider functionality or QoS). Besides, the execution time budget allocation is a trade-off after the mode change. Without understanding the application and differentiation of LO tasks according to their importance level, the proposed method can not guarantee the execution of relatively important LO tasks, which would avoid violation of functional requirements.

To mitigate such problems, an increasing number of papers advocate constructing systems that can provide essential services in the presence of failures (e.g., timing overruns) through graceful degradation. The authors of [70] and [72] highlighted the importance of graceful degradation for automotive

systems and demonstrated multiple approaches to degradation depending on the application and situation. They pointed out that designing a graceful degradation policy should be based on understanding the system's reaction to adverse events, but explicit mode transitions (i.e., once any of the HI tasks overruns, all LO tasks are discarded) are unnecessary. However, they did not show how to do this considering non-functional requirements – and our work fills that gap.

A finer-grained degradation strategy incorporating the notion of importance was proposed in [58]. Tasks are grouped as applications, and all LO criticality tasks from one application share the same importance value and are dropped simultaneously. Instead of dropping all LO tasks immediately, they propose dropping them when they have to. With the help of offline sensitivity analysis, the run-time policy controls the degradation of LO criticality applications based on the extent of a system overrun. However, one application can consist of several sub-functions, and each of them could have a different impact on the output and can even propagate interference to other applications when the sub-function is shared. One typical example from a real autonomous system is that *Visual Inertial Odometry* (VIO) can be regarded as one task in the task chain of the *Visual-based Simultaneous Localisation and Mapping* (vSLAM) application. It can provide pose estimation (transition and rotation) on its own and also can be fused with the result from LIDAR-based SLAM and GPS-based methods to provide a more accurate location estimate. Ideally, the degradation policy would choose to discard the successors of VIO in the vSLAM task chain but not the whole application for a better overall system utility and QoS – but this requires task-level, not application-level task dropping. In [26], the authors further emphasise that importance can provide fine-grained run-time graceful degradation. However, the proposed method still uses application-level task dropping, a limitation our work addresses.

Task dependency analysis can be introduced from the functional perspective to overcome this limitation. The method proposed in [52] attempts to assess the impact of a combination of faults and determine the task replication number, which can only enhance fault tolerance by providing a backup mechanism that helps prevent critical failures without criticality differentiation to ensure a certain utility function value in a fault scenario. However, the system utility model and task dependency analysis are based on the execution behaviour in the hardware platforms (e.g., probability of a processor

failure) for fault tolerance without considering the impact on the QoS from the holistic functional solution perspective. More recent work proposes using Bayesian Belief Networks for situation-aware dynamic risk assessment and braking behaviour prediction to model the probabilistic relationships between different tasks under different driving conditions [107]. The example illustrates the feasibility of using Bayesian Belief Networks in representing autonomous vehicles' performance based on probability propagation between tasks. However, the authors only focused on a single function which consists of tasks with dependencies (e.g., braking). The interference propagation between functions is not analysed. Furthermore, there has been little work on non-functionally degraded scheduling designs that take into account different system operating conditions and the propagation of functional failures after dropping specific tasks.

In summary, most research considers the functional and non-functional domains to be independent, even though there are undoubtedly influences between them. Therefore, in this thesis, we propose a novel inference-based degradation order definition method which incorporates awareness of the operational environment, e.g., the driving scenario. Besides, the proposed scheduling strategy can be assessed using functional criteria, thereby making our graceful degradation procedure feasible from the non-functional domain and also reasonable from the functional perspective.

2.8 Network Bandwidth Scheduling

As emphasised in Chapter 1 and Section 2.1.2, the research on resource scheduling should be extended from the task level to the network level because, for an autonomous system, the data transmission between different domains is vital to guarantee the safety of the system. Same with functions, data also have different criticality levels. For example, messages directly related to safety are regarded as safety-critical traffic (ST), and their transmission should be guaranteed. Non-safety-critical traffic can be classified into Class A and Class B. Class A includes the flows from non-safety critical traffics but can provide mission-critical services. Class B consists of the remaining traffic with even lower importance [83]. Figure 2.7 shows examples of streams in a typical autonomous driving system and AVB flows can be classified into Class A and Class B based on their importance in different solutions designed by system

engineers.

	Stream Info	Type	Source	Destination	Size (Byte)	Bandwidth Allocation (bps)
1	Navigation/AVM	ST	Infotainment	ADAS	60	704 k
2	Navigation data	AVB	Infotainment	HUD	2	80 M
3	Navigation/Audio	AVB	Infotainment	Cluster	2	48 k
4	Audio	AVB	Infotainment	AMP	11	125 k
5	Entertainment	AVB	Infotainment	RSE	1250	125 k
6	Entertainment	AVB	RSE	Rear Left Monitor	1250	80 M
	Entertainment	AVB	RSE	Rear Right Monitor	1250	80 M
7	GPS/V2X	AVB	Integrated Antenna	Infotainment	16	1 M
8	Control data	ST	Body	PT/Chassis	4	3 k
9	PT/Chassis info	ST	PT/Chassis	ADAS Sensor fusion	230	184 k
10	ADAS sensor fusion	AVB	ADAS Sensor fusion	Cluster	2	125 k
11	ADAS sensor fusion	AVB	ADAS Sensor fusion	HUD	2	125 k
12	Forward camera	AVB	ADAS Sensor fusion	Integrated Antenna	250	80 M
13	Control data	ST	ADAS Sensor fusion	ESC	625	500 k
14	Control data	ST	ADAS Sensor fusion	MDPS	625	500 k
15	AVM image	AVB	ADAS Sensor fusion	Display	1250	80 M
16	Mirrorless image	AVB	ADAS Sensor fusion	Mirrorless Display	1250	80 M

Figure 2.7: Example of streams in a typical autonomous driving system [83].

Modern automotive systems are implemented as distributed systems, connecting various ECUs with sensors and actuators, all with high bandwidth requirements. The sensors, including high-resolution camera images and high-density LiDAR point clouds, have particularly high bandwidth requirements. The traditional in-vehicle communication network is based on the controller area network (CAN) bus, which only has a bandwidth of 0.5 Mbit/s. The advantages of Ethernet-based networking are its low cost and high speed. It has a bandwidth of 100 Mbit/s or even more, satisfying the communication requirements of ADAS and even ADS [113]. Conventional Ethernet networks operate by broadcasting the message from a source node to all connected nodes in the network, with no regard for destinations. This means that only one node can transmit messages at any given time. Switched Ethernet addresses this issue by only sending traffic from sources to their destinations. This allows Ethernet switches to have multiple concurrent transmissions, greatly increasing the switches' efficiency. As the Ethernet switch can control the flows on the network, end-to-end delays throughout networks can be bounded. However, automotive communication consists of distinct services that are strictly

different in quality and potentially interfere with a flat common network. Standard switched Ethernet must be extended beyond simple traffic prioritization to provide real-time guarantees. A lack of deterministic behaviour leads to unpredictability in the communication system. Therefore, real-time Ethernet solutions, particular transport protocols focused on delivering low latency, and deterministic typically introduce delay guarantee.

Time-Sensitive Network (TSN)

The Ethernet-based in-vehicle network has been standardized in the IEEE 802.1 time-sensitive network (TSN) group since 2006 [57]. TSN standards enable the simultaneous transmission of different traffic classes, from best-effort (non-critical with the lowest priority) to deterministic real-time traffic (time-critical with the highest priority) [111], within a single network technology. This is a significant advantage to the current state-of-the-art, where systems frequently install independent networks for different applications [114]. Some prior works have verified that the traffic for autonomous driving satisfies the TSN transmission requirements in the in-vehicle network (IVN) and can be useful to reduce the End-to-End overall delay [83]. Figure 2.8 depicts a simplified example architecture of the IVN.

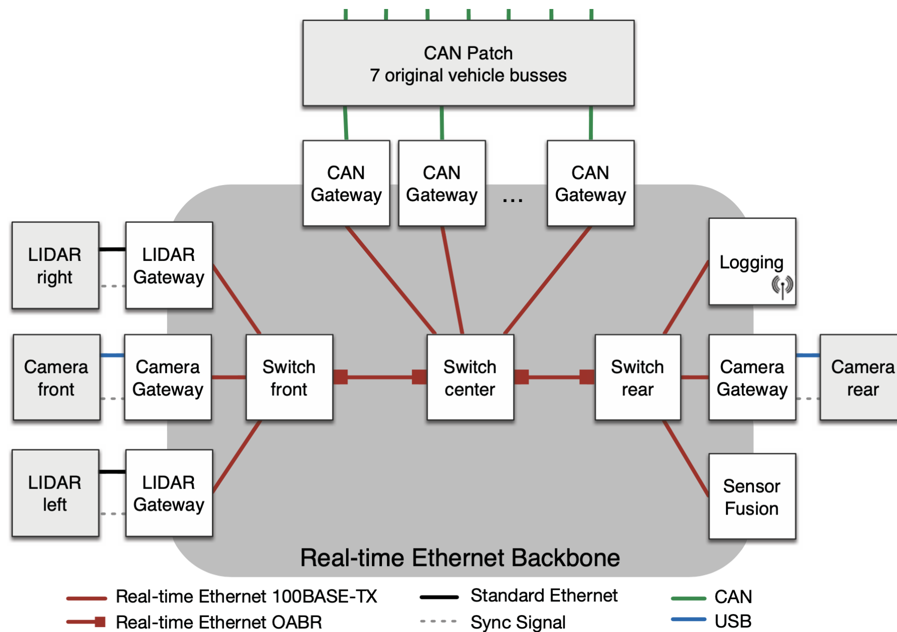

Figure 2.8: The simplified architecture of in-vehicle network [113].

In a TSN network, nodes will typically be switches and end stations. A stream (or flow) is a periodic multicast data transmission from one talker (the sender) to one or multiple listeners (the receivers). The senders and the receivers will be end stations, while switches will operate as forwarding nodes. While the stream defines the overall end-to-end communication between sender and receivers, the concept of a frame (also called a frame instance) identifies a particular message communicated between any two nodes. The period of the frame is equal to the period of the stream, while the length of the frame is calculated based on the data size of the stream and the link speed.

The transmission of TSN flow is based on a time schedule to guarantee time predictability. The IEEE 802.1AS-rev standard specifies the protocol for establishing and maintaining time synchronization between network devices. And also ensure the time schedule is followed correctly throughout the network. As mentioned before, the traffic classes of the autonomous system are different and can be encoded in the data link frame with a three-bit priority code point (PCP). Each TSN switch can queue up to eight distinct traffic classes with different priorities on every port. The IEEE 802.1Qbv standard defines scheduling mechanisms. The Time-Aware Shaper (TAS) determine when different traffic classes should be allowed to transmit on a network. TAS schedules consist of equal discrete time slots. Each slot specifies which gate should be open or closed at that specific time to ensure determinism and real-time. The schedule can be represented as a Gate Control List (GCL), which contains the time-slot entries and the associated configuration of gate states (GS).

Time-triggered switching provides full control of the timing of each frame in a switch. Frame memory management in a switch would need a major re-design as the communication schedule could require arbitrary frame reordering when forwarding frames. Figure 2.9 depicts the queue-based model of a TSN switch. Each queue is assigned to a gate, which has at any time two states open or close. When the respective gate is open, frames can be selected for transmission on the directed edge associated with the queue in first-in-first-out (FIFO) order. If the gate of a respective queue is closed, frames from this queue are not selected. The state of a gate may change from open to close and vice versa from close to open. These state changes are statically scheduled with respect to a synchronised time and defined at the design time of the network. The scheduling problem in TSN time-triggered communication is

finding the points in time for the opening and closing of the gates. It realises transmission with predictable delays [36].

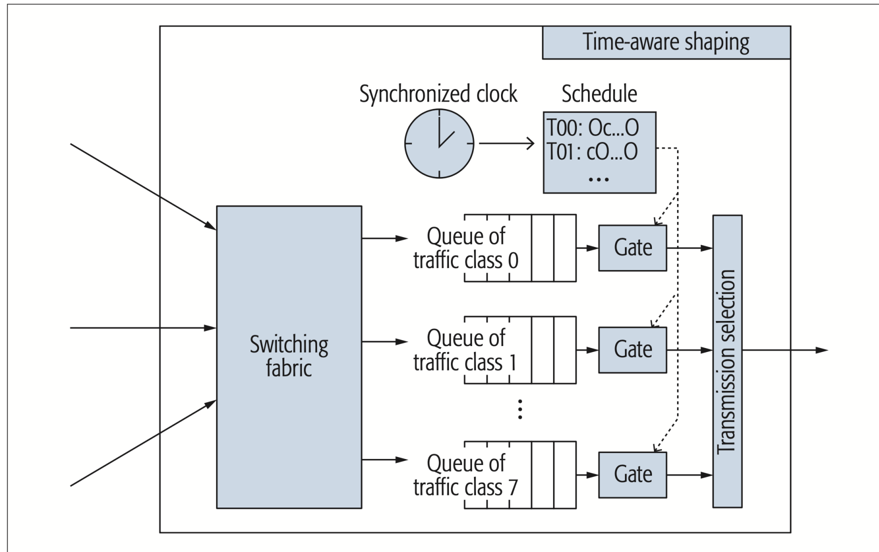

Figure 2.9: Overview of a TSN switch [114].

Several works (e.g., [100], [49]) propose methods for formalizing the combination of schedules for GCLs and procedures to generate the schedules for TSN based on satisfiability modulo theories (SMT). In [44], the authors proposed a fixed-priority scheduling (FPS) approach and developed a frame-level response time analysis with arbitrary deadlines, which can be used to verify the deterministic transmission of TSN. For ST frames, the consequences of missing time constraints can range from the degradation of service to potentially fatal system failure [33]. Therefore, it is vital to develop a fault tolerance capability for TSN. However, most current work [7], [69], [48] focuses on spatial redundancy, which is insufficient to handle temporal failures, including faults caused by incorrect arrival times. Therefore, the main purpose of this work is to improve the capability of TSN in terms of temporal fault tolerance.

In order to improve TSN's efficiency, preemption mechanisms could be utilised. Most existing systems are based on non-preemptable fixed-priority scheduling so that any low-priority frame could block the transmission of a high-priority frame for, at most, the length of one frame. If frame sizes are large, as is typical in ADAS and ADS, then the emergency (event-triggered) traffic could be blocked for a non-negligible period of time [71] in the ab-

sence of preemption. For example, the maximum valid Ethernet frame, i.e., 1542 bytes, could lead to a $12.336\mu s$ blocking time in a network with a 1G bit/s link rate [71]. That is unacceptable in automotive systems where the latency requirements can be as low as 5 μs [79]. It has been verified that the adoption of the preemption mechanism would not degrade traffic transmission performance [25], [117]. However, the existing standard only supports one level of preemption [111]. In [99], the authors identified this limitation and investigated the feasibility of multiple preemption levels. They demonstrated the necessity of multi-level preemption and introduced the technical details. However, they did not provide any solution to further illustrate the efficiency improvement in tackling other issues. The most recent work [10] combined multi-level preemption support with the credit-based shaper algorithm and proved that the maximum blocking delay caused by lower-priority traffic could be reduced considerably. Besides, their strategy dramatically decreases the response time of frames with higher priority. However, the drawback caused by credit-based shaper has not been completely resolved.

2.9 Summary

This chapter comprehensively introduces essential background knowledge, including scheduling algorithms and formal analysis methods. The software and hardware complexity highly motivates our research to prove time-predictable and resource-efficient scheduling methods that guarantee task execution and data transmission to satisfy criticality-dependent timing requirements. Mixed-criticality scheduling methods are discussed, which are fundamental steps inspiring us to find more efficient ways to tackle timing uncertainties. Especially when facing timing faults of safety-critical tasks, improving system resilience, survivability of non-safety-critical tasks, and maintaining systems' functional quality of service (QoS) is critical. Graceful degradation is an efficient method that can be designed carefully to provide more fine-grained controllability to release resources, tolerate timing faults, and maintain system operation safely with high resource utilisation efficiency. However, considering both functional and non-functional requirements is essential for task scheduling design, especially for autonomous system functions, because their system-wide functionality impact levels are different and cannot be evaluated based on metrics designed from a plain non-functional point of view. Task-dependent

analysis-based static scheduling methods provide temporal isolation for tasks with varying levels of criticality. This can reduce resource waste caused by hardware isolation, which requires more hardware resources and easily leads to utilisation unbalances. However, when facing system mode changes, static scheduling methods bring challenges and provide a different viewpoint about realising task-level degradation. Finally, task-level scheduling research needs to be extended to network bandwidth resources. List-based scheduling approach (i.e., GCL table) is involved in our work to realise temporal isolation for safety-critical traffic transmission, which is similar to the static schedule for task execution and has been introduced in the state-of-the-art TSN technology. The server-based method can be involved in designing scheduling strategies to manage remaining bandwidth, which can further provide temporal isolation for the transmission of relatively more important non-safety-critical traffic and with awareness of fairness for rest traffic transmission, avoiding prolonged channel block. The contributions of this thesis are based on these ideas and methods. In the following few chapters, details of these contributions will be introduced and discussed in detail.

Chapter 3

Context- and Causality-aware Graceful Degradation for Mixed-Criticality Scheduling

Autonomous systems are of high complexity and often regarded as mixed-criticality systems (MCS) in which functions are allocated criticality levels according to risk assessment based on safety standards. Typically, tasks have different real-time requirements across criticality levels, and the estimated worst-case execution times (WCETs) are distinct. Further, limitations in computational resources increase the difficulty of integrating tasks onto one shared hardware platform. Conventionally, all non-safety critical tasks must be discarded or suspended to guarantee the execution of safety-critical tasks when facing a timing fault. This typically leads to a considerable decrease in the system's Quality-of-Service (QoS). Achieving more graceful degradation is critical to minimizing QoS reduction.

This chapter focuses on tackling timing faults and proposes a novel graceful degradation strategy for use in a mixed-criticality context. Thus, when a system has multiple operational modes depending on the environment or an operational task, our approach can provide an effective way of managing degradation to maximize QoS, which is currently not sufficiently recognized in MCS. Furthermore, the proposed causality analysis-based degradation process "bridges the gap" so functional dependencies are considered in scheduling design and thus leads to a graceful degradation that is both feasible and reasonable in functional and non-functional terms. The evaluations show that

QoS can be better preserved using the proposed context-aware degradation process when compared with more conventional MCS scheduling approaches.

3.1 Introduction

In the last decade, the development of computational platforms and the dramatic advances in artificial intelligence (AI) and machine learning (ML) have enabled autonomous systems to operate without human supervision, even completing some demanding missions with the ability to adapt to the environment. On the other hand, there have been several safety reports of fatal accidents with autonomous vehicles (AVs) [90]. Evidently, we are still facing significant challenges in autonomous systems before providing real safety benefits to society.

As introduced in Chapter 1, these safety challenges can be considered from both functional and non-functional perspectives. Functionally, safety assurance processes need to be developed to assist in promoting the safety and reliability of functions in the system. Therefore, some standards that incorporate functional safety and Risk Assessment (RA) (e.g., ISO 26262, ISO/PAS 21448, etc.) have been proposed to provide guidance on safety concepts within the automotive industry. Further, they can be used to assign different criticality values (e.g., Automotive Safety and Integrity Levels (ASIL)) to functions in the system [41]. For non-functional issues (i.e., timing faults, as the focus of this work), the scheduling policies should be developed in a way that ensures the system complies with timing constraints; hence tasks can be executed correctly and at the right time [63]. However, the increasing number of functions that are required to run on a single resource-limited hardware platform raises the difficulty of functional integration, especially when tasks have different criticality levels and various real-time requirements. Furthermore, the difference in worst-case execution time estimates (WCETs) under different system criticality modes makes scheduling even more complicated.

For complex mixed-criticality systems (MCS), the underlying research problem can be interpreted as how to ameliorate the conflict of partitioning (w.r.t. ASILs) for safety assurance and the effectiveness of shared resource usage. Fundamentally, the task scheduling strategy should guarantee the execution of safety-critical functions (e.g., brake control) without interference from any non-safety-critical tasks (e.g., entertainment system) and tolerate a timing

fault (overrun) of any safety-critical function. Conventional scheduling strategies change the system mode once a timing fault occurs in any safety-critical (HI criticality) task. After that, all non-safety-critical (LO criticality) tasks are suspended or dropped, and only safety-critical tasks are allowed to run to their maximum estimated WCET [31]. However, as Burns *et al.* observed [32], a task considered as LO criticality may still contain mission-critical functions and be vital for correct system operation. Such tasks should be maintained as long as possible and should not be suspended or dropped unless their execution prevents the safety-critical tasks from completing successfully.

Therefore, improving the survivability of those LO criticality tasks is vital, which can be realised by providing more refined graceful degradation. However, in the real-time systems community, graceful degradation methods are usually based on a simple characterization of the functional domain, where the degradation order is assumed to be defined according to the known importance of each task. Besides, the importance level is also simply determined based on the predefined criticality level by system designers, which may be insufficient due to the limited number of levels defined by current standards (e.g., the five ASILs QM & A-D). One attempt to mitigate Quality-of-Service (QoS) reduction is through application-level task dropping (i.e., all LO tasks in the same application are discarded simultaneously), which is commonly used because of the difficulty of identifying the relative importance among LO tasks from an individual application [58]. However, that still leads to unnecessary system QoS decreases.

The research problem becomes even more complicated when considering multi-modal sensing. The criticality of the sensors and sensor processing can depend on the environment, e.g., some sensors are more effective in fog and rain, or on the situation, e.g., proximity sensors such as ultrasound are more useful in low-speed (parking) manoeuvres. That means the system can exhibit multiple operational modes depending on the environment or operational task. System designers can adapt the solution to deal with such mode changes. However, that is rarely considered by a task scheduler. Instead, it is assumed the functional solution and the importance order of tasks are unchanged. As a result, some LO tasks would be dropped first in the new operational mode, although they could be more useful and important than the remaining tasks. The unnecessary QoS reduction may be countered with adaptive changes to the importance order. Although such scheduling strategies can guarantee system

safety, they can still lead to unnecessary degradation of QoS, i.e., the lack of understanding of functional dependencies would lead to a non-optimal graceful degradation in scheduling with respect to the functional domain.

In this chapter, we attempt to effectively deal with mode changes caused by overrunning safety-critical (HI-criticality) tasks while maximizing QoS through graceful degradation of LO-criticality tasks with the awareness of system-wide functionality and operational modes (e.g., due to the change of environmental conditions). Our approach comprises two fundamental steps.

First, we present a method to formally define the relationship between tasks and characterize dependencies, followed by determining a metric to evaluate the functional QoS. Based on these fundamental steps, we propose a novel and general strategy, *Context-aware Task-level Graceful Degradation* (CTGD), for mixed-criticality scheduling to address the above issues. To illustrate the superiority of our proposed method, we selected the state-of-the-art application-level task dropping proposed in paper [58] as the baseline method. (i.e., if a LO-criticality task must be dropped, all LO-criticality tasks from the same application should be dropped simultaneously.) The evaluation results prove that with the awareness of multiple operational modes, causality analysis-based task-level graceful degradation can preserve more LO-criticality tasks at each task dropping point and maximize system-wide functional Quality of Service (QoS).

The **main contributions** of this work are:

- We address the issues of application-level task dropping and the lack of awareness of the task's influence in the system during importance order definition. Task-level importance ordering is done by introducing a description of functional dependencies, which is utilized to determine the importance value of each task in the system, which is more granular than simply using ASILs.

- We consider the environmental conditions and their impacts on the system, which is referred to as *the context* in this work. In different contexts, the reliability and thus importance order of LO-criticality tasks could be different. Instead of using one single importance order for graceful degradation in the scheduling design, our proposed method can ensure that the system is running with higher functional QoS for each context/scenario.

- Based on the functional dependency and failure propagation analysis, a controlled degradation process then discards computational load at the task level rather than simultaneously dropping all *LO*-criticality tasks in an application. Thus, our method can improve survivability and keep more *LO*-criticality but mission-critical tasks during overload, even under high utilization situations.

- In addition to extensive experiments on synthetic tasks, we demonstrate the importance order definition through an example that intuitively illustrates its rationality and feasibility.

The rest of the chapter is organized as follows: In Section 3.2, the proposed method (CTGD) is detailed, followed by the formulation of proposed implementation strategies in Section 3.3 and Section 3.4 from functional and non-functional perspectives, respectively. The evaluation is introduced in Section 3.5. Finally, we conclude the work with a discussion on future work in Section 3.6.

3.2 Method Overview

An autonomous system comprises a limited number of applications, each consisting of several tasks. This Section will thoroughly introduce the system model used by the proposed CTGD from functional and non-functional perspectives. For the functional model, task dependency is involved in describing the functional relationship and used to analyse the influence propagation (i.e., how the state change of one node affects the failure probabilities of the rest of the nodes in the network). Each function is a unit that can be verified independently and also a discrete execution unit with the same meaning as a task in the following. For the non-functional model, all tasks are released simultaneously at run-time, and their execution can preempt or be preempted based on predefined priority. We adopt a fixed-priority scheduler (FPS) to manage task execution, which is used by most commercial real-time operating systems and mandated by automotive industry standards and guidelines [1].

3.2.1 System Model from Functional Perspective

The proposed novel graceful degradation method considers system-wide influence propagation to ensure that decisions to discard tasks minimise the

reduction in the system's functional QoS. To achieve this, we have three fundamental steps to follow to build a functional system model:

- Understand the task dependency in the system;
- Define the relationship and influence propagation;
- Determine the metric for QoS evaluation.

Step 1: Understand the task dependency in the system with the awareness of multiple operational modes: Inevitably, autonomous functions are implemented based on environmental and state perceptions. Thus, the availability of robust sensing information and algorithms is crucial for all robotic and autonomous platforms. Except for the complexity of task dependencies, as identified by Gadd et al. [61], harsh weather and lighting conditions pose non-trivial challenges to the development of autonomous systems, especially for traditional sensing systems, forcing us to face the problem of multi-modal sensing and multiple operational modes depending on the environment or operational task. Current state-of-the-art sensing methods can help to perceive the operational environment. Such functions are referred to as *context-aware tasks* in this work, helping us identify the operational mode to make the proposed method flexible when facing a dynamically changing environment.

As an example, Figure 3.1 depicts the end-to-end pipeline of one of the potential solutions from our delivery mobile robot (AAIP robot), which is developing as one safety case to support the research about safety operation concept [101]. It is not difficult to notice that data transmission exists between different tasks, which implies failure propagation and is regarded as influence propagation in this work. It is worth noting that, except for tasks with data transmission (dependency), the system has many independent functions.

We have adopted the Robot Operating System (ROS) to integrate all the algorithms illustrated in Figure 3.1. ROS provides a wide range of drivers and state-of-the-art algorithms, along with powerful developer tools. It serves as a set of software libraries and tools for building robot applications, making it convenient for the development of robotics projects [89]. The lack of support for real-time systems has been addressed in the creation of ROS2, which is why we have chosen ROS2 as the algorithm integration tool.

In ROS2, a callback is considered the minimal schedulable entity, with five different types (i.e., timer, subscription, service, client, and waitable call-

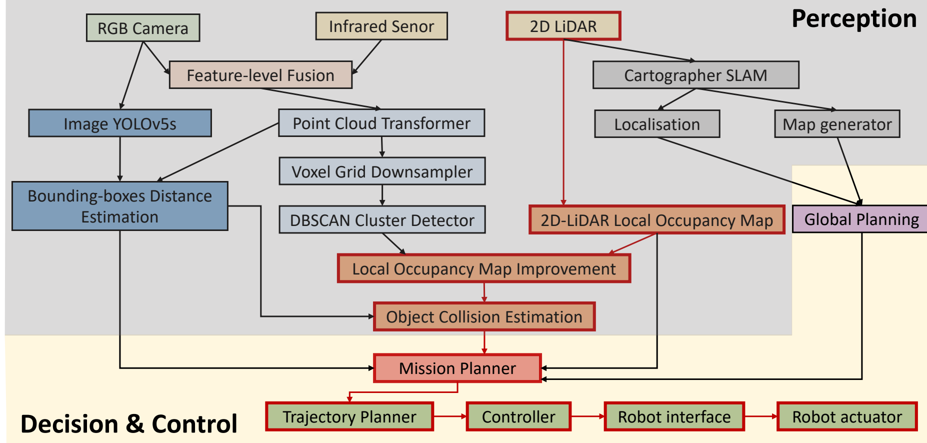

Figure 3.1: AAIP mobile robot end-to-end pipeline.

backs). Message transmission between different tasks is achieved by implementing callback functions, which integrate publisher and subscriber mechanisms. This means that if a ROS node cannot generate results correctly or timely and be identified before publishing, the publisher can send a predefined message (e.g., "invalid results") to its successors, and the callbacks of its successors can perform corresponding operations. Additionally, if a node does not receive valid input results, its callback can also be designed to perform corresponding operations. In this work, if HI-criticality tasks overrun (i.e., the execution time of the callback exceeds the predefined time budget), the predefined LO-criticality task nodes can be skipped, or their callbacks can be set to only keep data refreshing without further processing.

Step 2: Define the relationship and influence propagation: As an essential step, the relationship between different system tasks needs to be formally defined. Based on understanding task dependency in autonomous systems, we use a directed acyclic graph (DAG) to represent task relationships. A DAG is a graph $G_{A_x}, A_x \in S$, which can be defined as $G_{A_x} = (V_{A_x}, E_{A_x})$. In this work, we assume that there are N applications in the system $S = \{A_0, A_1, ..., A_N\}$. Each application is composed of tasks (nodes) with dependency ($V_{A_x} = \{v_{x0}, v_{x1}, ..., v_{xm}\}$) and is represented by a relatively small DAG , making it easier to perform interference modeling and analysis from functional perspectives. The connections among application DAGs can construct a large-size system-level DAG that represents task dependencies between appli-

cations and also establishes potential interference propagation between tasks from different applications. It is worth noting that the system has many independent functions except for tasks with data transmission (dependency). Thus, tasks without dependency (independent tasks) are regarded as an application comprising only one task. E_{A_x} (arrows on each link) denotes the set of directed edges between the vertices V_{A_x} (nodes) from application A_x such that by following a path of vertices from one node to another along the direction of each edge, no path will revisit a vertex. A system consisting of multiple applications connected by a shared node can be represented by a larger graph, $G_S = (V_S, E_S)$, where $V_S = \bigcup_{\forall A_x \in S} V_{A_x}$ and $E_S = \bigcup_{\forall A_x \in S} E_{A_x} \bigcup E_c$. E_c denotes the set of edges connecting nodes from different applications.

Based on the structure of the task graph, we also need to find a method to define the functional relationship between tasks formally. Discrete Bayesian Belief Networks (BBNs) are adopted as they can represent a set of variables and their conditional dependencies via a DAG, which is able to support the formulation of mathematical metrics for further evaluation [16]. In [118], an example of the real-world application of BBN-based system representation is provided. The BBN of a system S can be denoted as $\mathcal{N}_f = (\mathcal{X}_S, G_S, P_s)$. $\mathcal{X}_S$ represent the variables. Each node (task) v in G_S corresponds one-to-one with a discrete random variable $x_v \in \mathcal{X}_S$ with a finite set of mutually exclusive states. The directed links $E_S \subseteq V_S \times V_S$ of G_S specify assumptions of conditional dependence (and independence if no links) between random variables according to the d-separation criterion [80]. P_s denotes a set of conditional probability distributions $P(X_v | X_{par}(v)) \in P_s$, for each variable $x_v \in \mathcal{X}_S$. $X_{par}(v)$ denotes the set of parent variables of node $v \in V_S$ and also known as the conditioning variables of x_v.

According to the definition of the chain rule [80], a BBN represents the joint distribution over all the variables represented in the DAG as Equation (3.1) shows, and the marginal and the conditional probabilities can be computed for each node of the network.

$$P(\mathcal{X}_S) = \prod_{v \in V_S} P(x_v \mid X_{par(v)}) \tag{3.1}$$

When considering context-aware nodes, they can work as *prior* information for related tasks to initialise their conditional probability tables (CPTs). Thus, as this work requires, BBN is highly compatible with supporting multiple operational modes. As one essential characteristic of BBN, inference can

be used to represent the influence propagation, i.e., the state change of any variable in the system can lead to the variation of failure probabilities of variables directly and indirectly related to the changed variable. As described in [16], given a distribution $P(x_1, ..., x_n)$, the inference is the process of computing functions of the distribution. Marginal inference can be an example concerned with the computation of the distribution of a subset of variables, possibly conditioned on another subset. For example, given a joint distribution $P(x1, x2, x3, x4, x5)$, a marginal inference given evidence approximation is as Equation (3.2):

$$P(x_5 \mid x_1 = true) \propto \sum_{x_2, x_3, x_4} P(x_1 = true, x_2, x_3, x_4, x_5) \tag{3.2}$$

Independent tasks can be included in the joint distribution of one system, and relevant terms can be eliminated during the marginal distributions' approximation. As a result, we can use a formal way to prove that these independent tasks have no system-wide functional influence propagation.

BBN parameters training: The system's functions can be segmented into discrete units for verification based on risk assessment. Therefore, we assume that each unit's state can be well-defined, and the structure of the system's BBN model is known and realised by system designers. Let's take the mobile delivery robot as an example. Assume we only consider two driving conditions (context), i.e., C_0 = well-lit condition, C_1 = badly-lit environment. The camera can work as a lighting condition-related context-aware task and highlight the decreased confidence in camera-based functions. For the RGB-object detection task (variable), we determine two states (i.e., correct and incorrect). Correct means generating true-positive and true-negative results, while incorrect represents generating false-positive and false-negative results. Under different lighting conditions C_i, we can collect the rates of true-positive (correct) and false-positive (incorrect) and false-negative (incorrect). The collected data can be used to train the conditional probability table of the RGB-object detection variable. It is not difficult to know the required parent of each verifiable task based on task dependency analysis when extended to the system level. Then, in a similar way, relevant data for each task can be collected during the long-time system test under different possible conditions defined by its parents. In this work, we assume that the data has been collected and conditional probability tables have been trained. Then, the conditional prob-

ability table for each verifiable unit (task) integration can work as an initial state and can be continuously improved during system tests (run-time observation). Some research has started to build BBN for an autonomous system, as an example paper [107] uses data on observations of real-world (highway) driving to construct BBNs.

Step 3: Determine the metric for QoS evaluation: The research problem is to define the decision-making procedure to determine the dropped task at each step which results in the minimum interference with the behaviour of the system. To solve this problem, we need to build the criteria to describe the QoS, which should be maximised during degradation. We use utility theory to determine the metric for QoS, and the objective of decision-making is to identify the options that produce the highest expected utility. In probabilistic networks, calculating each variable's marginal distribution can reflect the influence propagation between variables. This means that a change in the state of any one variable in the system will induce changes in the marginal distribution of the variable that are directly and indirectly related to the changed variable. To save computation, we can also assume a subset of all task nodes as key nodes ($\mathcal{X}_{key}$), where their execution will significantly impact the system's performance. Therefore, the marginal distribution of each key node is used to construct the expected utility function (EU) as Equation (3.3) shows.

$$EU = \prod_{x_i \in \mathcal{X}_{key}} P(x_i) \sum_{x_i \in \mathcal{X}_{key}} u(x_i) \tag{3.3}$$

The decision variable, $\mathcal{D}$, consists of all "undropped" (i.e., still active) LO criticality tasks in the system, grouped into subsets according to different applications $\mathcal{D} = \{[\mathcal{D}_{1,1}, ..., \mathcal{D}_{1,n}], ..., [\mathcal{D}_{m,1}, ..., \mathcal{D}_{m,k}]\}$, where $\mathcal{D}_{m,n}$ represents the nth LO task from an application m. The decision options (which task to drop) are mutually exclusive. Based on the trained BBN model, task discarding means removing a variable, and the tables of its successors need to be updated. Then, the system can be described using the updated conditional probability tables. Section 3.3.2 uses a detailed example to explain the table updating procedure.

The variable x_i belongs to the set $\mathcal{X}_{key}$ (i.e., $x_i \in \mathcal{X}_{key} \subset \mathcal{X}_S$). The marginal distribution $P(x_i)$ of each key node (variable) x_i can be calculated when making a decision (i.e., task removal, updating relevant tables). This will impact the choice of alternative at $\mathcal{D}$ because the change in the system's

conditional probability table can lead to differences in expected utility. $u(x_i)$ represents a local utility function attached to the key node to reflect the decision's local impact. The definition of the local utility function may need to consider the impact within each application. The sum of them is worked as a coefficient to the probability term and can be used to amplify the differences and impact the final EU value, providing a more sensitive QoS measure. In this work, we only adopt the probability term because a more detailed design involving such factors should be carefully discussed and verified. The definition of local utility will be discussed in future work; here, we consider it as a constant value.

3.2.2 System Model from Non-functional Perspective

In autonomous systems, tasks have different levels of criticality, forming an MCS. Such systems can run at different levels/modes (e.g., safety-critical, mission-critical and non-critical). In each mode, the tasks that must be guaranteed can be classified according to their allocated criticality. Appropriate scheduling policies should be adopted to guarantee that the tasks satisfy their timing requirements. WCET is one of the vital parameters, and its estimation may be different under different system modes based on their safety requirements — a higher criticality mode would typically be associated with a more pessimistic WCET estimate — and directly influences the schedulability of the system. For example, tasks are initially executed in the LO criticality mode in a dual criticality system. Once a fault (overrun) occurs in any HI criticality task, the system criticality mode changes from LO to HI, after which all LO criticality tasks are suspended or dropped and only HI criticality tasks are remained to run to their maximum estimated WCETs [31]. For systems with multiple criticality modes, the system designer needs to map tasks to several criticality-level groups according to safety requirements [74].

As the first work to bridge the gap between functional and non-functional domains, we focus on improving the performance w.r.t. functional QoS when tackling non-functional problems. To limit the uncertainty and complexity introduced by the non-functional factors, we adopt the standard dual-criticality model introduced in [120] and assume all tasks can be assigned with an implicit deadline according to the predefined period. For time-triggered tasks, the relative deadline equals their period. For event-triggered tasks, the period and the relative deadline are determined by their minimum inter-arrival time. Based on

this assumption, in a dual-criticality system, each task τ_i is defined by a tuple $\tau_i := \Big(T_i, D_i, C_i(LO), C_i(HI), Li\Big)$, with the period or minimum inter-arrival time T_i, deadline D_i, the execution time of low-criticality $C_i(LO)$, and of high-criticality $C_i(HI)$ and an individual criticality level L_i. τ_i is synonymous with x_i, it is a vertex in the system, and its variable is represented by x_i in a BBN. The tasks are mapped into LO and HI criticality groups. The tasks that must be guaranteed in any circumstance are defined as HI-criticality tasks. Others are regarded as LO-criticality, which can be sacrificed to protect the execution of HI-criticality tasks. Moreover, we consider criticality-dependent WCETs. Thus, each LO-criticality task has only one estimated WCET $C(LO)$ (i.e., if L_i is LO, $C_i(HI)$ is omitted), and each HI-criticality task has two WCET estimates: $C(LO)$ and $C(HI)$ ($C(LO) \leq C(HI)$).

As introduced at the beginning of Section 3.2, for the purpose of analysis, we assume that all tasks are released simultaneously at run-time and adopt an FPS to manage task execution. Therefore, the priority definition is vital and also plays the bridge role in introducing the functional factor to the non-functional domain because, in this work, the importance is determined according to the impact level on the system's functional QoS. The importance order guided priority definition can provide execution preference to tasks that have a greater impact on the system. The detailed formulation will be introduced in Section 3.3.

Please note that in this part of the work, we do not consider changes in the system mode. A context-aware task can signal a change in the driving condition, requiring an adjustment to the importance order, but it does not signify a change in the system mode itself (e.g., LO system mode to HI system mode). Consequently, we do not require task schedulability analysis for this purpose. To ensure the safe operation of the system, we can identify an available idle time slot to modify the priority of tasks (since adjusting the importance order implies changing the priority of "Low Importance" tasks to optimize the operation). Furthermore, in the event of an overrun, we can utilize the importance order that corresponds to the current priority to mitigate risks, such as potential deadlocks arising from priority conflicts.

3.2.3 Survivability and Graceful Degradation

The critical target of MCS is to guarantee the execution of tasks required in different system modes to ensure the system's safety. Conventionally, the

scheduling strategy for a dual-criticality system is dropping all LO-criticality tasks immediately when any HI-criticality task faces a timing fault (i.e., overrun in this work). However, as introduced in [58], when a HI task overruns its $C(LO)$, it is likely that it will not execute up to its $C(HI)$. It is more likely that the task might only overrun by a small margin, and there exists sufficient slack time to support LO tasks execution. Moreover, some LO criticality tasks are regarded as mission-critical functions, and simply dropping them is unnecessary. Therefore, it is becoming increasingly evident that we need to maintain the functions as long as possible, even though they are defined as LO tasks, and the system is executing in a higher-criticality mode.

In this work, we introduce a graceful degradation strategy to enhance the survivability of LO-criticality tasks and also expect to maintain the QoS to improve the resilience of the system. That means we need to provide a higher degree of controllability and granularity over how a system degrades. The fundamental step for graceful degradation strategy design is to determine the degradation order for droppable tasks. Similar to the strategy in [58], the degradation order is defined by importance order, and task dropping starts from the least important task according to the extent of the overrun of HI-criticality tasks. During an overrun, it is possible that the system could move into the HI-criticality mode while maintaining all, or at least some, of its LO-criticality tasks if there is sufficient slack left in the system. As soon as an overrun reaches the point at which a LO-criticality task must be dropped, the task with the lowest importance is suspended/discarded. The fine-grained task level discarding can delay the system-level mode change where all LO-criticality tasks are discarded. Once the HI tasks execution is back to normal, the fewer LO tasks need to be reinstalled. Thus, faster system performance can be recovered — this is regarded as another advantage of the proposed degradation. The overrun severity and relevant dropped tasks are calculated offline and recorded in memory. The detailed procedure will be introduced in Section 3.4. At run-time, we only have to monitor the actual execution time of HI-criticality tasks and discard LO tasks according to the fixed rule, thus with limited overheads.

3.2.4 The Algorithm Pseudo-code

The pseudo-code in Algorithm 1 illustrates the main pipeline of CTGD. Initially, we assume the BBN $P(\mathcal{X}_S)$ has been well-trained for the analysed system

and $\mathcal{X}_S$ is the set of all variables. The LO and HI task sets are known and denoted by Γ_{LO}, Γ_{HI}, respectively. $\mathcal{S}^{LO}$ is a set consisting of all applications with droppable LO tasks (i.e. $\mathcal{S}^{LO} = \bigcup_{i=0}^{n} A_i, \exists v \in V_{A_i} \bigcap \Gamma_{LO}$). $V_{A_i}^{LO}$ denotes the set of LO tasks from application A_i (i.e., $V_{A_i} = \bigcup_{j=0}^{m} v_{ij}, \forall v_{ij} \in V_{A_i} \bigcap \Gamma_{LO}$). V_S^{LO} is the set comprising all application task sets with LO tasks (i.e., $V_S^{LO} = \bigcup_{\forall A_i \in \mathcal{S}^{LO}} V_{A_i}$).

Algorithm 1: The main workflow of CTGD

Input: $P(\mathcal{X}_S), \Gamma_{LO}, \Gamma_{HI}, \mathcal{X}_S, \mathcal{S}^{LO}, V_S^{LO}, V_{A_i}^{LO}$

Output: $\mathcal{T}_{C_i}, \forall C_i \in \{C_0, C_1, \cdots, C_n\}$

1 **for** C_i *in* $\{C_0, C_1, \cdots, C_n\}$ **do**

2 $\quad N_{C_i} \leftarrow \Gamma_{LO}$;

3 $\quad \mathcal{T}_{C_i} : \mathcal{O} \mapsto \emptyset$;

4 $\quad P_{C_i}(\mathcal{X}_S) = P(\mathcal{X}_S | C = C_i)$;

5 $\quad \mathcal{I}_{C_i} = \text{ImportanceOrder}(P_{C_i}(\mathcal{X}_S), \Gamma_{LO}, \Gamma_{HI}, \mathcal{X}_S, \ \mathcal{S}^{LO}, V_S^{LO}, V_{A_i}^{LO})$;

6 $\quad \mathcal{P}_{C_i} = \text{PriorityAssignment}(\mathcal{I}_{C_i})$;

7 $\quad \mathcal{T}_{C_i} = \text{Offline-SensitivityAnalysis}(N_{C_i}, \Gamma_{HI}, \Gamma_{LO}, \ \mathcal{O}, \widetilde{\mathcal{O}}, \mathcal{I}_{C_i})$;

8 **end**

Based on the offline-sensitivity analysis, the discarded tasks should be calculated offline for different system modes according to the severity of HI-criticality tasks' overrun ($\mathcal{O}$) and the importance order of LO tasks, respectively (i.e., $\mathcal{T}_{C_i}, \forall C_i \in \{C_0, C_1, \cdots, C_n\}$). The system modes serve as priors for updating the associated conditional probability tables and updating the initial state of the system's BBN for importance order definition, as lines 4 and 5 of the pseudo-code show. Then the importance order-based priority allocation (line 6) is prepared for the sensitivity analysis (line 7), and $\mathcal{T}_{C_i}$ can be calculated iteratively. The while loop will be stopped either all LO tasks are dropped, or the overrun severity is larger than the predefined threshold ($\widetilde{\mathcal{O}}$). For each system mode, N_{C_i} is the set of undropped LO-criticality tasks that need to be initialised by all LO tasks. The pseudo-codes of *ImportanceOrder()*, *PriorityAssignment()*, and *Offline-SensitivityAnalysis()* are given later in Algorithms 2-4.

In summary, we propose a method that considers the relationship between different tasks and also different applications in the system to lay the foundation for finer-grained degradation control. The influence of each task dropping will be reflected in the probabilistic task graph and used to formulate the cri-

teria to determine the degradation order (the mathematical formulation will be introduced in Section 3.3). With the awareness of the environmental situation, e.g., the driving condition, which is determined as *prior* information to the probabilistic graph model, different degradation orders (i.e., importance orders) can make the system run with relatively high functional QoS under different driving circumstances.

3.3 Formulation of Importance Ordering

As the first step of the proposed CTGD strategy, this section details the importance ordering, which is intended to improve the resilience of systems based on understanding the holistic functional dependency.

3.3.1 Allocation of Importance Orders

The importance allocation defines the degradation order of *LO* criticality tasks during timing overruns. The influence of discarding a task can propagate network-widely as the tasks would have inter-dependencies. Thus, it could also contribute to other applications. The metric defined by expected utility introduced in Section 3.2.1 can be used to evaluate the impact level of each *LO*-criticality task. The simplified expected utility function can be reformed as Equation (3.4) and is essentially the multiplication of the marginal probabilities of key nodes.

$$EU = \prod_{x_i \in \mathcal{X}_{key}} P(x_i) \tag{3.4}$$

We adopt the sum-product algorithm to compute the marginals of all key nodes as Equation (3.5) shows.

$$P(x_i) = \sum_{\mathcal{X}_S \setminus x_i} P(\mathcal{X}_S) \tag{3.5}$$

The proposed method can provide different graceful degradation orders for different driving conditions to provide more flexibility and maintain the system QoS with the awareness of context. As introduced in Section 3.2.3, the driving condition works as *prior* to initialise the BBN work, and the joint distribution can be updated by $P_{C_j}(\mathcal{X}_S) = P(\mathcal{X}_S|C = C_j)$, and j represent the mode index. Based on the joint distribution of the specific mode, we can get the corresponding importance order of *LO*-criticality tasks $\mathcal{I}^{\mathcal{LO}}{}_{C_j}$ according to

the proposed $ImportanceOrder$ Algorithms. During the importance order calculation, if one or more tasks $\mathcal{X}_d = \{x_1, x_2, \ldots, x_k\}$ from the droppable LO-criticality task set Γ_{LO} is/are selected to be discarded, the chosen task is regarded as generating false output, and its state will be set as incorrect and removed from the network, which works as *prior* information to its successors' CPTs (i.e., related conditional probability tables are updated). Then, the joint distribution of the system needs to be updated as $\hat{P}_{C_j}(\mathcal{X}_S) = P_{C_j}(\mathcal{X}_S \setminus \mathcal{X}_d)$. Based on the latest joint distribution, we can calculate the utility of the system without the dropped tasks according to Equation (3.4). The pseudo-code of Algorithm 2 introduces the procedure of importance order definition. We determine the application level discarding order first (from lines 1 to 12) and then decide the task-dropping order for each application (from lines 13 to 33). In that way, the graceful degradation can start from the task in relatively less important applications and make our method more practical.

The selected application or task to be dropped should maintain the maximum expected utility value (with the lowest impact on the system's service) and is assigned the lowest importance. The problem can be defined as:

$$\max \left(\prod_{x_i \in \mathcal{X}_{key}} \sum_{\mathcal{X}_S \setminus x_i \setminus \mathcal{X}_d^m} \hat{P}_{C_j}(\mathcal{X}_S) \right), \forall \mathcal{X}_d^m \in \hat{\mathcal{X}} \tag{3.6}$$

For application discarding order definition, $\mathcal{X}_d^m$ represents the set of LO tasks from application A_m (i.e., $\mathcal{X}_d^m = \mathcal{X}_{V_{A_m}}, V_{A_m} = \bigcup_{j=1}^n v_{mj}, \forall v_{mj} \in \Gamma_{A_m} \bigcap \Gamma_{LO}$, and $\hat{\mathcal{X}} = \bigcup_{\forall A_m \in S} \mathcal{X}_d^m$). For relative task discarding order define in one application, $\mathcal{X}_d^m$ represents the selected task τ_m and $\hat{\mathcal{X}}$ is the set of LO tasks from the analyzed application A_k (i.e., $\mathcal{X}_d^m = x_{v_m}, \hat{\mathcal{X}} = \bigcup_{m=0}^n x_{v_m}, \forall v_m \in \Gamma_{A_k} \bigcap \Gamma_{LO}$). Once the dropped application or task is selected, both it and the related edges will be removed from the graph. The system's joint distribution will be updated, and the network will be considered as a new one with a new structure, as lines 12 and 30 show. Then, the next test round can be launched. The procedure will be repeated until all LO critical tasks are removed and the importance order under driving condition C_i is defined and denoted as $\mathcal{I}_{C_i}^{LO}$.

As aforementioned, the importance order guided priority definition can provide execution preference to relatively important tasks. In this work, we assume that all tasks from HI-criticality task set Γ_{HI} share the same importance level and higher than all LO-criticality tasks. The importance order of HI tasks is denoted by $\mathcal{I}_{C_i}^{HI}$. Then, the Importance order of all tasks in the system

$\mathcal{I}_{C_i}$ in specific driving conditions can be obtained through $\mathcal{I}_{C_i} \leftarrow \mathcal{I}_{C_i}^{LO} \bigcup \mathcal{I}_{C_i}^{HI}$. The first element from list $\mathcal{I}_{C_i}$ has the lowest importance level.

3.3.2 Example of BBN-based Importance Order Definition

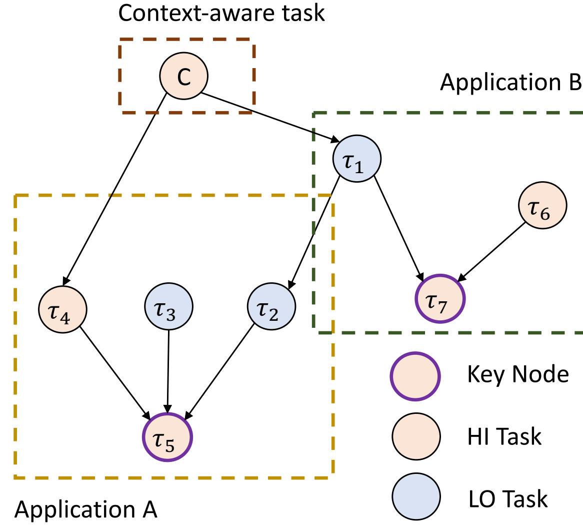

Figure 3.2: The example of a system represented by a Bayesian network.

To help to understand, we use one example to demonstrate the BBN-based context-aware importance ordering. Figure 3.2 illustrates a system comprising two applications, A and B. Each application consists of several tasks with dependencies. The edge between task τ_1 and τ_2 builds the connection between the two applications. Task C denotes the context-aware node that works as *prior* information of the related tasks' CPTs (i.e., τ_1 and τ_4), and its state can be either normal (C_0) or abnormal (C_1). Figure 3.3 shows the randomly generated CPTs for each task (node). In each table, the condition variables represent the predecessors of the corresponding variable (task). The state of driving context C can be used to initialise the table of τ_1 and τ_4. Assuming the current driving condition is abnormal, i.e., $C = C_1$, the tables of τ_1 and τ_4 are updated as Figure 3.4 shows. Then, the element C can be eliminated, and the initialisation of the Bayesian network under abnormal mode is finished. Then, we start to calculate the marginal probability of τ_5 and τ_7, respectively, following Equation (3.5).

The application-level degradation order is first determined then the relative discarding order is determined within each application. During application-

Algorithm 2: The ImportanceOrder Algorithm

Input: $P_{C_i}(\mathcal{X}_S) = P(\mathcal{X}_S | C = C_i), \Gamma_{LO}, \Gamma_{HI}, \mathcal{X}_S,$ $\mathcal{S}^{LO}, V_S^{LO}, V_{A_i}^{LO}$

Output: $\mathcal{I}_{C_i}$

/* Decide Application-level Discarding Order: */

1 $\hat{\mathcal{S}} \leftarrow \mathcal{S}^{LO}$;

2 $\mathcal{D}_{app} = \emptyset$;

3 $\hat{P}_{C_i}^{app}(\mathcal{X}_S) \leftarrow P_{C_i}(\mathcal{X}_S)$;

4 **while** $\hat{\mathcal{S}} \neq \emptyset$ **do**

5   **for** A_i *in* $\hat{\mathcal{S}}$ **do**

6     $\hat{P}_{C_i}^{A_i}(\mathcal{X}_S) = \hat{P}_{C_i}^{app}(\mathcal{X}_S \setminus \mathcal{X}_{V_{A_i}})$; $EU^{A_i} = \prod_{x_i \in \mathcal{X}_{key}} \hat{P}_{C_i}^{A_i}(x_i)$;

7   **end**

8   $ind = max(EU^{A_i})$ /* application with the lowest impact */

9   $\mathcal{D}_{app} \leftarrow A_{ind}$;

10   $\hat{\mathcal{S}} \leftarrow \mathcal{S}^{LO} \setminus A_{ind}$;

11   $\hat{P}_{C_i}^{app}(\mathcal{X}_S) \leftarrow \hat{P}_{C_i}^{app}(\mathcal{X}_S \setminus \mathcal{X}_{V_{A_{ind}}})$;

12 **end**

/* Decide Task-level Discarding Order: */

13 $\hat{V}_S \leftarrow V_S^{LO}$;

14 $\mathcal{I}_{C_i}^{LO} = \emptyset$;

15 $\hat{P}_{C_i}^{\tau}(\mathcal{X}_S) \leftarrow P_{C_i}(\mathcal{X}_S)$;

16 **while** $\hat{V}_S \neq \emptyset$ **do**

17   **for** A_i *in* $\mathcal{D}_{app}$ **do**

18     $\hat{V}_A \leftarrow V_{A_i}^{LO}$;

19     $\mathcal{D}_{\tau} = \emptyset$;

20   **while** $\hat{V}_A \neq \emptyset$ **do**

21   **for** v_j *in* $\hat{V}_A$ **do**

22         $\hat{P}_{C_i}^{v_j}(\mathcal{X}_S) = \hat{P}_{C_i}^{\tau}(\mathcal{X}_S \setminus x_{v_j})$; $EU^{v_j} = \prod_{x_i \in \mathcal{X}_{key}} P_{C_i}^{v_j}(x_i)$;

23   **end**

24       $ind = max(EU^{v_j})$/* task with the lowest impact */

25       $\mathcal{D}_{\tau} \leftarrow \tau_{ind}$;

26       $\hat{V}_A \leftarrow V_S^{LO} \setminus \tau_{ind}$;

27       $\hat{V}_S \leftarrow V_S^{LO} \setminus \tau_{ind}$;

28       $\hat{P}_i^{\tau}(\mathcal{X}_S) \leftarrow \hat{P}_{C_i}^{\tau}(\mathcal{X}_S \setminus x_{v_j})$;

29   **end**

30     $\mathcal{I}_{C_i}^{LO} \leftarrow \mathcal{D}_{\tau}$;

31   **end**

32 **end**

33 $\mathcal{I}_{C_i} \leftarrow \mathcal{I}_{C_i}^{LO} \bigcup \mathcal{I}_{C_i}^{HI}$;

C is priori

C_0	Normal
C_1	Abnormal

$P(\tau_6)$

$\tau_6 = 0$	0.80
$\tau_6 = 1$	0.20

$P(\tau_3)$

$\tau_3 = 0$	0.70
$\tau_3 = 1$	0.30

$P(\tau_2|\tau_1)$

$\tau_1 = 0$	$\tau_2 = 0$	0.85
$\tau_1 = 0$	$\tau_2 = 1$	0.15
$\tau_1 = 1$	$\tau_2 = 0$	0.01
$\tau_1 = 1$	$\tau_2 = 1$	0.99

0: Correct
1: Incorrect

$P(\tau_1| C)$

C_0	$\tau_1 = 0$	0.85
C_0	$\tau_1 = 1$	0.15
C_1	$\tau_1 = 0$	0.30
C_1	$\tau_1 = 1$	0.70

$P(\tau_4|C)$

C_0	$\tau_4 = 0$	0.75
C_0	$\tau_4 = 1$	0.25
C_1	$\tau_4 = 0$	0.85
C_1	$\tau_4 = 1$	0.15

$P(\tau_7|\tau_1, \tau_6)$

$\tau_1 = 0$	$\tau_6 = 0$	$\tau_7 = 0$	0.90
$\tau_1 = 0$	$\tau_6 = 0$	$\tau_7 = 1$	0.10
$\tau_1 = 0$	$\tau_6 = 1$	$\tau_7 = 0$	0.30
$\tau_1 = 0$	$\tau_6 = 1$	$\tau_7 = 1$	0.70
$\tau_1 = 1$	$\tau_6 = 0$	$\tau_7 = 0$	0.85
$\tau_1 = 1$	$\tau_6 = 0$	$\tau_7 = 1$	0.15
$\tau_1 = 1$	$\tau_6 = 1$	$\tau_7 = 0$	0.05
$\tau_1 = 1$	$\tau_6 = 1$	$\tau_7 = 1$	0.95

$P(\tau_5|\tau_2, \tau_3, \tau_4)$

$\tau_4 = 0$	$\tau_3 = 0$	$\tau_2 = 0$	$\tau_5 = 0$	0.90
$\tau_4 = 0$	$\tau_3 = 0$	$\tau_2 = 0$	$\tau_5 = 1$	0.10
$\tau_4 = 0$	$\tau_3 = 0$	$\tau_2 = 1$	$\tau_5 = 0$	0.85
$\tau_4 = 0$	$\tau_3 = 0$	$\tau_2 = 1$	$\tau_5 = 1$	0.15
$\tau_4 = 0$	$\tau_3 = 1$	$\tau_2 = 0$	$\tau_5 = 0$	0.80
$\tau_4 = 0$	$\tau_3 = 1$	$\tau_2 = 0$	$\tau_5 = 1$	0.20
$\tau_4 = 0$	$\tau_3 = 1$	$\tau_2 = 1$	$\tau_5 = 0$	0.75
$\tau_4 = 0$	$\tau_3 = 1$	$\tau_2 = 1$	$\tau_5 = 1$	0.25
$\tau_4 = 1$	$\tau_3 = 0$	$\tau_2 = 0$	$\tau_5 = 0$	0.40
$\tau_4 = 1$	$\tau_3 = 0$	$\tau_2 = 0$	$\tau_5 = 1$	0.60
$\tau_4 = 1$	$\tau_3 = 0$	$\tau_2 = 1$	$\tau_5 = 0$	0.35
$\tau_4 = 1$	$\tau_3 = 0$	$\tau_2 = 1$	$\tau_5 = 1$	0.65
$\tau_4 = 1$	$\tau_3 = 1$	$\tau_2 = 0$	$\tau_5 = 0$	0.10
$\tau_4 = 1$	$\tau_3 = 1$	$\tau_2 = 0$	$\tau_5 = 1$	0.90
$\tau_4 = 1$	$\tau_3 = 1$	$\tau_2 = 1$	$\tau_5 = 0$	0.05
$\tau_4 = 1$	$\tau_3 = 1$	$\tau_2 = 1$	$\tau_5 = 1$	0.95

Figure 3.3: The conditional probability table of the system.

level calculation, all *LO* tasks of the dropped application will be discarded simultaneously. The marginal probability of each key node is calculated to determine the utility function (i.e., QoS) of a system based on Equation (3.5). Application-dropping possibilities can be used to find out the application-level dropping order. In this example, we have two possibilities: (1) discarding all *LO* tasks in application A first, and (2) dropping all *LO* tasks in application B first. Take the first possibility as an example. τ_2 and τ_3 would be dropped. Their states would be set to 1 (i.e., they cannot generate correct results). Based on this possibility (i.e., condition), the tables (i.e $P(\tau_5, \tau_4)$) of their successors τ_5 can be updated as Figure 3.4 shows.

After finishing graph updating, the marginal probability of each key node τ_5 and τ_7 should be found. First of all, the marginal probability should be formulated based on the possibility. τ_5 is taken as an example, and its marginal probability based on the possibility (1) is shown in Equation (3.7).

$$P(\tau_5) \propto \sum_{\tau_4} P(\tau_4)P(\tau_5, \tau_4)\times$$

$P(\tau_4)$

$\tau_4 = 0$	0.85
$\tau_4 = 1$	0.15

$P(\tau_1)$

$\tau_1 = 0$	0.30
$\tau_1 = 1$	0.70

$P(\tau_5, \tau_4) = P(\tau_5|\tau_4)$

$\tau_4 = 0$	$\tau_5 = 0$	0.75
$\tau_4 = 0$	$\tau_5 = 1$	0.25
$\tau_4 = 1$	$\tau_5 = 0$	0.05
$\tau_4 = 1$	$\tau_5 = 1$	0.95

Figure 3.4: The updated tables of τ_1, τ_4 and τ_5.

$$\sum_{\tau_1,\tau_6,\tau_7} P(\tau_1)P(\tau_6)P(\tau_7,\tau_6,\tau_1) \tag{3.7}$$

Evidently, τ_5 only has a relationship with τ_4, and the equation can be further simplified as Equation (3.8) shows. In the same way, the marginal probability of τ_7 can be simplified by Equation (3.9).

$$P(\tau_5) \propto \sum_{\tau_4} P(\tau_4)P(\tau_5,\tau_4) \tag{3.8}$$

$$P(\tau_7) \propto \sum_{\tau_1,\tau_6} P(\tau_1)P(\tau_6)P(\tau_7,\tau_6,\tau_1) \tag{3.9}$$

Based on the marginal probability formulation of each key node and the tables attached to related nodes, the sum-product method is used to eliminate the elements one after another in the formulation equation (e.g., Equation (3.8)) to get the final marginal probability values of each key node. For instance, the marginal probability calculation of τ_5 only needs to eliminate one element, τ_4 and is taken as an example to demonstrate the calculation procedure. As Figure 3.5 illuminates, one intermediate factor ψ represented by Equation (3.10) would be generated to assist in τ_4 elimination following Equation (3.11) and the marginal probability of $P(\tau_5)$ is proportional to it. Following the same procedure, the marginal probability of τ_7 can be calculated (i.e., $P(\tau_7 = 0) \propto 0.717$ and $P(\tau_7 = 1) \propto 0.283$).

$$\psi(\tau_4,\tau_5) = P(\tau_4)P(\tau_5,\tau_4) \tag{3.10}$$

$$P(\tau_5) \propto \sum_{\tau_4} \psi(\tau_4,\tau_5) \tag{3.11}$$

Finally, the utility value of possibility (1) can be obtained by $EU_1 = P(\tau_5 = 0) \times P(\tau_7 = 0) = 0.645 \times 0.717 \approx 0.462$ according to Equation (3.4).

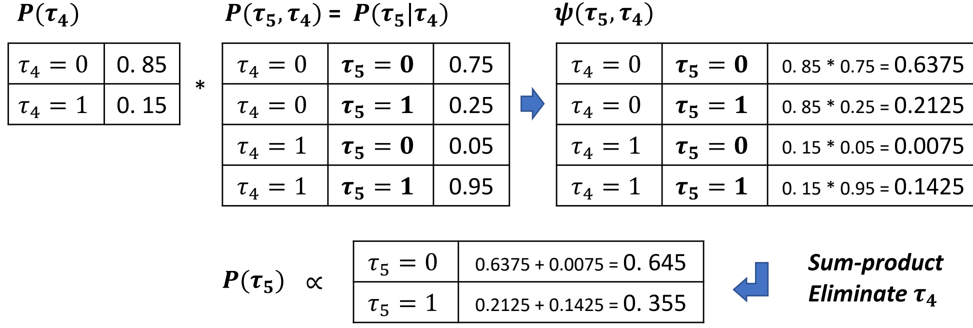

Figure 3.5: The example of τ_4 sum-product based elimination.

Following the same calculation procedure, the utility value of EU_2 under the possibility (2) is approximately 0.508. Because $EU_2 > EU_1$, the LO task τ_1 in application B should be discarded first. From now on, we will start to determine the relative order within each application. In this example, there is only one LO-criticality task in application B, and it is also the first task in the final degradation order list with the lowest importance value. For the LO tasks from application A, the task with the highest utility value after discarding should be determined as the one with the lowest importance value in the specific application. In this example, τ_2 should be discarded first with utility value 0.527, which is larger than the expected value (0.483) of dropping τ_3 first. Suppose there are more LO tasks in application B. The conditional probability tables should be updated after fixing the first dropped task, and the procedure would be repeated until the task order is determined. In this example, the final degradation order is determined as: (τ_1, τ_2, τ_3).

3.4 Formulation of Graceful Degradation

The importance order definition method introduced in the last section focuses on functional aspects. This section will introduce the non-functional perspective formulation of the proposed CTGD method. The task dropping points (i.e., the severity of HI tasks' overrun) for LO-criticality tasks should be found offline, which defines a graceful degradation procedure based on the importance order and the severity of the timing fault (overrun) at run time.

3.4.1 Priority Assignment

In this work, we assume all HI criticality tasks share the same importance value higher than all LO criticality tasks. Context-aware tasks are classified into the HI-criticality task set, and it is not necessary to differentiate them from other HI tasks for priority assignment. The relative importance order of LO tasks is determined based on the method introduced in the last subsection. After the importance order of all tasks is confirmed (i.e., $\mathcal{I}_{C_i}$). We adopt Audsley's priority assignment technique-based method to assign priority to each task, which attempts to allocate lower-importance tasks with a lower priority. The pseudo-code is illustrated in Algorithm 3.

This approach works as follows [58]: For each priority level, beginning at the lowest. Based on the schedulability analysis, we can check the schedulability of each task at this level. If more than one task is schedulable (i.e., $Sched = True$), assign priority to the task with the lowest importance value. In the pseudo-code, p_l is the range of priority in the system and equals the number of tasks in the system (i.e., $len(\mathcal{I}_{C_i})$. The priority allocated to each task τ is denoted by $\mathcal{P}_{C_i}(\tau)$.

Schedulability analysis is essential to verify that the execution of tasks can satisfy their predefined deadline in the worst-case scenario. *Adaptive Mixed-Criticality* (AMC) analysis is widely used and recognised as the most effective approach and has become the standard approach for fixed priority-based MCSs, which has been thoroughly introduced in Section 2.5.2. In this work, we adopt the *AMC-rtb* [24] method for schedulability analysis. However, the response time analysis could be a pessimistic estimation and reduce the schedulability of the task set. The reason is that jobs of LO-critical tasks may not execute for the entire busy period of a HI-criticality task in the LO mode. After the mode change point, there are no newly released LO jobs. Besides, for HI-critical tasks, jobs released before the mode change point only contribute no more than $C(LO)$ value. AMC-rbt-based response time analysis is pessimistic but computationally efficient. There exists a trade-off between accuracy and computing fast, which is out of the scope of this work.

3.4.2 Sensitivity Analysis and Graceful Degradation

Once the system mode change (from LO to HI) is triggered, we do not drop all LO-critical tasks immediately but start from the least important task ac-

Algorithm 3: The PriorityAssignment with importance

Input: $\mathcal{I}_{C_i}$

Output: $\mathcal{P}_{C_i}$

1 Initialise all tasks with the highest priority level;
2 $p_l = len(\mathcal{I}_{C_i})$;
3 $\mathcal{P}_{C_i} : \tau \mapsto P_l, \forall \tau \in \mathcal{I}_{C_i}$;
4 **for** l *in* $Rang(p_l)$ **do**
5     $Candidate = \emptyset$;
6   **for** τ *in* $\mathcal{I}_{C_i}$ **do**
7         $\mathcal{P}_{C_i}(\tau) = l$;
8         $Sched = \text{SchedulabilityAnalysis}(\mathcal{I}_{C_i}, \mathcal{P}_{C_i}(\tau))$;
9   **if** $Sched = True$ **then**
10             $Candidate \leftarrow \tau$;
11   **end**
12   **else**
13   Continue;
14   **end**
15   **end**
16   Allocate the lowest priority level to the task with the lowest importance (I) task
17     $ind = min_I(Candidate)$;
18     $\mathcal{P}_{C_i}(\tau_{ind}) = l$;
19     $\mathcal{I}_{C_i} \leftarrow \mathcal{I}_{C_i} \setminus \tau_{ind}$;
20 **end**

cording to the extent of the overrun of HI-criticality tasks.

Sensitivity Analysis

Offline sensitivity analysis is used to determine the extent of the overrun required to drop a task with a particular level of importance. The procedure consists of the following steps:

1. Find out unschedulable points during increasing overrun of HI-criticality tasks.

2. Record the overrun extent and dropped LO-criticality task(s), then increase the extent of the overrun to search for the next dropping point.

3. Repeat the process until all possible dropping points are recorded.

As the pseudo-code for Algorithm 4 shows, all HI-criticality tasks' $C(LO)$ WCET values are assumed to be increased by the same percentage $\Delta_{\mathcal{O}}$ during sensitivity analysis (from line 10 to 13). At each round, the response time of each LO task under the latest schedulable state needs to be backed up (from lines 7 to 9). With increasing the $C(LO)$ of HI-criticality tasks until any of the following requirements, $R_i(LO) \leq D_i$ and $R_i^* \leq D_i$, are not satisfied (line 15), which can be calculated according to Equation (3.12) and (3.13). Then the backed-up response time $\hat{R_d}(LO)$ of dropped task τ_d is denoted as φ_{τ_d}, which is linked to the specific overrun (task dropping point, i.e., $\hat{\mathcal{O}}$) through the 2D array $\mathcal{R}_{\tau_d \hat{\mathcal{O}}}$. This array with be initialised as an empty set (line 3). The dropped task τ_d will be recorded into a $\Psi_{dropped}$ set.

One or more LO tasks should be discarded during sensitivity analysis at each dropping point. That means their interference can be eliminated after the specific time point (e.g., φ_{τ_d}). Therefore, the system is regarded as still working in LO mode. Then, we attempt to increase the overrun to find the next task dropping point, and Equation (3.12) can be used to estimate the response time. The third term of Equation (3.12) represents that after some specific time points, the already dropped LO tasks will not impact the execution of any tasks, even with a higher priority.

$$\begin{aligned} R_i(LO) &= C_i(LO) \\ &+ \sum_{\tau_j \in hp(i) \wedge \tau_j \notin \Psi_{dropped}} \left\lceil \frac{R_i(LO)}{T_j} \right\rceil \cdot C_j(LO) \\ &+ \sum_{\tau_j \in hp(i) \wedge \tau_j \in \Psi_{dropped}} \left\lceil \frac{\varphi_j}{T_j} \right\rceil \cdot C_j(LO) \end{aligned} \tag{3.12}$$

The response time analysis during mode change is based on Equation (3.13), which can dominate the response time analysis in HI mode:

$$\begin{aligned} R_i^* &= C_i(HI) \\ &+ \sum_{\tau_j \in hpL(i) \wedge \tau_j \notin \Psi_{dropped}} \left\lceil \frac{R_i(LO)}{T_j} \right\rceil \cdot C_j(LO) \\ &+ \sum_{\tau_j \in hpL(i) \wedge \tau_j \in \Psi_{dropped}} \left\lceil \frac{\varphi_j}{T_j} \right\rceil \cdot C_j(LO) \\ &+ \sum_{\tau_k \in hpH(i)} \left\lceil \frac{R_i^*}{T_k} \right\rceil \cdot C_k(HI) \end{aligned} \tag{3.13}$$

where R_i^* represents the response time of analysed task τ_i running with the $WCET$ in HI mode (i.e., $C_i(HI)$); $hpL(i)$ is the set of LO-critical tasks with a priority higher than or equal to the analysed task, and $hpH(i)$ denotes the set of HI-critical tasks with a priority higher than or equal to the analysed task. If the system mode change event impacts task τ_i, then the mode change time point should be earlier than $R_i(LO)$. Equation (3.13) provides the response time upper-bound of the analysed task, which includes the interference from LO tasks during mode change. Based on Equation (3.12), we can provide the bound of the possible interference of non-dropped LO tasks presented by the second term. The third term of Equation (3.13) has the same meaning as the third term of Equation (3.12).

Graceful Degradation

The dropping points and corresponding discarded tasks (i.e., $\mathcal{T}_{C_i}(\mathcal{O})$) considering conditions are defined for graceful degradation based on offline sensitivity analysis and fixed. If the overrun continues to increase, the calculation process is repeated until any stopping requirements are met. During runtime, we only need to monitor the actual execution time of HI-criticality tasks. If

Algorithm 4: The Offline-SensitivityAnalysis

Input: $N_{C_i}, \Gamma_{HI}, \Gamma_{LO}, \mathcal{O}, \widetilde{\mathcal{O}}, \mathcal{I}_{C_i}$

Output: $\mathcal{T}_{C_i}, \forall C_i \in \{C_0, C_1\}$

```
1  𝒪 ← 0%;
2  𝒯_{C_i} : 𝒪 ← ∅;
3  ℛ_{τ_0 𝒪_{0%}, τ_1 𝒪_{0%}, ..., τ_n 𝒪_{100%}} ← ∅;
4  Ψ_dropped ← ∅;
5  Δ_𝒪 = 10%;
6  C_τ^orig(LO) = C_τ(LO);
7  while N_{C_i} ≠ ∅ & 𝒪 ≤ 𝒪̃ do
8      for τ_i in Γ_LO do
9          R̂_i = R_i(LO);
10     end
11     𝒪 ← 𝒪 + Δ_𝒪;
12     for τ in Γ_HI do
13         C_τ(LO) ← min{C_τ^orig(LO)(1 + 𝒪), C_τ(HI)};
14     end
15     for τ_i in 𝓘_{C_i} do
16         while R_i(LO) > D_i or R_i^* > D_i do
17             for τ_d in 𝓘_{C_i} ⋂ N_{C_i} do
18                 𝒪̂ = 𝒪 − Δ_𝒪;
19                 𝒯_{C_i}(𝒪̂) ← τ_d;
20                 N_{C_i} ← N_{C_i} \ τ_d;
21                 φ_{τ_d} = R̂_{τ_d};
22                 ℛ_{τ_d 𝒪̂} = φ_{τ_d};
23                 Ψ_dropped ← τ_d;
24                 break;
25             end
26         end
27     end
28 end
```

any HI task overruns its LO WCET up to the recorded dropping point, the corresponding LO tasks can be directly suspended or dropped.

3.5 Evaluation

3.5.1 Experiment Setup

The evaluation is based on results produced using a simulation implemented in Python. Our simulator can randomly generate task dependencies based on constraints from real-world applications (e.g., the task number limitation and the width and depth of the task graph for each application). Each application is regarded as a set of tasks with dependencies. The context-aware task is a concept from a functional perspective. From a structural point of view, some source nodes from the randomly generated graph can be regarded as context-aware tasks. The criticality definition is determined based on the dependency between LO and HI task in the real world, which is introduced in [93]. Besides, the initial conditional probability tables are generated randomly according to the method introduced in [9]. The *experiment setup* has the following phases:

(1) Task dependency generation: an existing directed acyclic graph (DAG) generator [127] is used in simulating task graphs of systems in this work. Each DAG represents one application, and the maximum depth (the number of layers) is randomly chosen from 4 to 6. The number of generated nodes in each layer is uniformly distributed from 2 to 8. To bound the complexity for the following graph calculation, we assume that each system is composed of three applications and the task number of the whole system is between 20 and 35. Furthermore, one task from each application is randomly selected to simulate the task dependency between different applications. Under different system utilisations, we randomly generate ten different graph structures. Please note that the DAG is only involved in simulating the functional relationship between tasks in the system. As mentioned in Section 3.4.1, we adopt a fixed-priority scheduler according to Audsley's priority assignment [12] based method. Therefore, all tasks are assumed to be released simultaneously at runtime. The consideration of execution precedence constraints will be introduced in our future work.

(2) Execution time generation: After fixing the system graph structure, UUnifast [53] is used to synthesise the execution time and period of each task under different system utilisations. The relative deadline of each task is equal to its

period. Tasks from the same application are assumed to share the same period. Each LO criticality task is allocated a single execution time $C(LO)$. HI criticality tasks have dual execution time assignments; $C(LO)$ values are the same as for LO criticality tasks. In this work, we assume $C(HI) = 2 \cdot C(LO)$, which means the overrun severity ($\widetilde{\mathcal{O}}$) will never exceed 100%. Schedulability analysis is used to guarantee tasks' schedules when the system runs under LO mode, HI mode and during mode change.

(3) LO criticality task selection: The selection of LO tasks starts with selecting HI criticality tasks in the system. In the generated graph, the sink node of each application is considered as the fusion node, which should be set as HI criticality. At least one node from the parent set of any HI criticality task should be set as HI and is randomly selected. Following this rule, we define all HI criticality tasks, and the remaining tasks are classified into the LO criticality group.

(4) Conditional probability distribution generation: The previously generated task structure is fed into a CPT generator, which is published in [9]. The randomly generated CPTs are then attached to each task node in our system. Each node has only two states (correct and incorrect), and the definition of correct and incorrect can be found in Section 3.3.2. The context-aware task works as *prior* information for each system, and we assume the value is known as it will not impact the evaluation of our proposed method.

(5) Degradation order definition and sensitivity analysis: We first determine the application discarding order, and then the task dropping order for each application to define the degradation order. Our evaluation needs to illustrate how our method can enable more LO criticality tasks to survive during system overrun and maintain a relatively higher expected utility. Therefore, the expected utility (EU) and the number of remaining LO-criticality tasks should be recorded during offline sensitivity analysis. The difference (improved percentage) of EU ($\Delta EU_{\mathcal{O}}$) and survived task values ($\Delta S_{\mathcal{O}}$) are calculated as Equation (3.14) (3.15) and gathered to demonstrate the advantage of our method.

$$\Delta EU_{\mathcal{O}} = [EU^{\mathcal{O}}_{BBN} - EU^{\mathcal{O}}_{App})/EU^{\mathcal{O}}_{App}] \times 100\% \tag{3.14}$$

$$\Delta S_{\mathcal{O}} = [S^{\mathcal{O}}_{BBN} - S^{\mathcal{O}}_{App})/\mathcal{D}_{All}] \times 100\% \tag{3.15}$$

$EU^{\mathcal{O}}_{BBN}$ denotes the maintained EU at each task dropping point $\mathcal{O}$ with our

proposed BBN-based method. $U_{App}^{\mathcal{O}}$ represents the maintained value if we drop all LO criticality tasks from the same application simultaneously. Suppose the difference value $\Delta EU_{\mathcal{O}}$ is larger than zero. In that case, that means our approach can maintain the system at a higher QoS at each task-dropping point, at which one or more LO tasks should be discarded to guarantee the execution of HI criticality tasks. $S_{BBN}^{\mathcal{O}}$ denotes the number of remaining LO criticality tasks at each task dropping point with our method. $S_{App}^{\mathcal{O}}$ represents the number if we do application-level task dropping. $\mathcal{D}_{All}$ is the set of all droppable tasks in the analysed system.

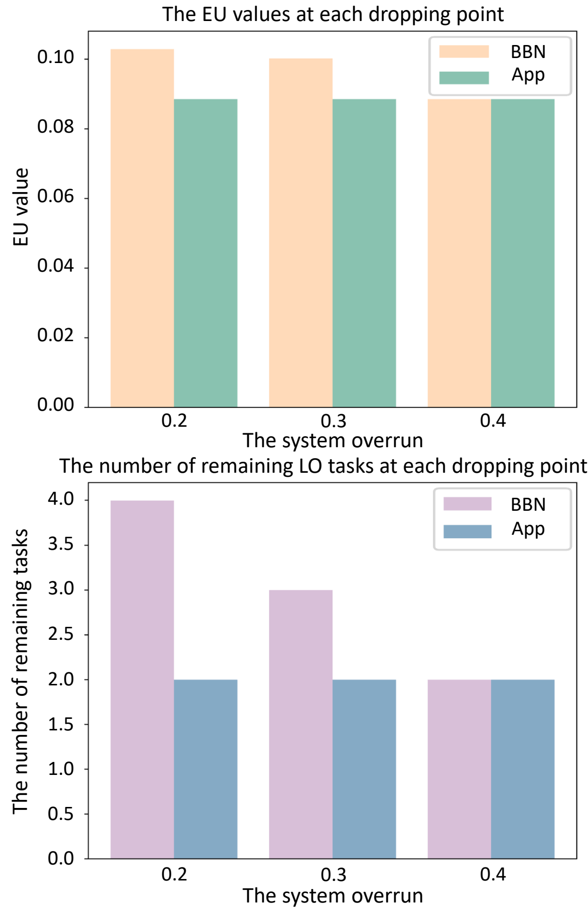

Figure 3.6: The EU value and remaining LO tasks at each task dropping point of a specific graph. (with Util = 0.7, 3 applications, 18 tasks, of which 6 are LO criticality tasks)

The difference value ΔS illustrates how many more tasks can survive during the system's overrun. During offline sensitivity analysis, different graph

structures with different CPTs, execution times, LO task sets will have different task-dropping points. As the example in Figure 3.6 shows, the task-dropping points calculated by offline sensitivity analysis are 0.2, 0.3 and 0.4. That means that one or more LO-criticality tasks should start to be discarded when the severity of HI-criticality tasks achieve these three points (i.e., 20%, 30%, and 40%), respectively. With the increase of the overrun further, the system has sufficient resources to support the execution of HI-criticality tasks and it is not necessary to discard more LO-criticality tasks. That means, based on both the application-level discarding method and our proposed method, two LO-criticality tasks will never be discarded. Compared to the application-level discarding method at the first task-dropping point, our strategy can maintain a higher EU. The fine-grained degradation control also results in enabling 50% more LO tasks to survive. Once the overrun reaches 20%, dropped LO tasks are from two different applications. The standard application discarding method dropped all LO tasks of both applications. However, the system still has enough slack. Two tasks are unnecessarily dropped. When the overrun increases to 0.3, our proposed method can maintain one more task, which will be dropped with a system overrun of 0.4. Evidently, our proposed method can keep LO tasks as long as possible. In the meanwhile, the EU value is also maintained.

To ensure that the experiment was of sufficient scale to give credible results, we synthesised task sets under 7 different utilisation ($U = 0.3, 0.4, ..., 0.9$) and randomly generated 10 different system structures (i.e., task dependency graph) for each workload. Under each workload with a different graph structure, 30 different conditional probability distributions (CPDs) were randomly generated. For each system with individual utilisation and conditional probability distribution, the different values of $\Delta EU_{\mathcal{O}}$ and $\Delta S_{\mathcal{O}}$ at each task dropping point for the overrun test will be calculated and the mean different value $\Delta\overline{EU}$ and $\Delta\overline{S}$ can be calculated according to Equations (3.16) and (3.17). $\mathcal{T}(\mathcal{O})$ is a set comprised of all task-dropping points (i.e., the severity of system overrun $\mathcal{O}$), identified by offline sensitivity analysis.

$$\Delta\overline{EU} = mean(\Delta EU_{\mathcal{O}}), \forall \mathcal{O} \in \mathcal{T}(\mathcal{O}) \quad (3.16)$$

$$\Delta\overline{S} = mean(\Delta S_{\mathcal{O}}), \forall \mathcal{O} \in \mathcal{T}(\mathcal{O}) \quad (3.17)$$

Overall, for each system utilisation, we use 300 different systems with either different task dependencies or different probabilistic relationships to

evaluate our proposed method. In order to compare the performance of a system based on the application level degradation and our proposed strategy, the difference value distributions and the mean values under different graph structures in specific system utilisation are collected. The results of systems with the utilisation of 0.6 and 0.9 are selected to illustrate the performance under medium and highly-loaded situations, respectively. Furthermore, the comprehensive results of 2100 different systems are collected and presented.

3.5.2 Extensive Experiments with Synthetic Tasks

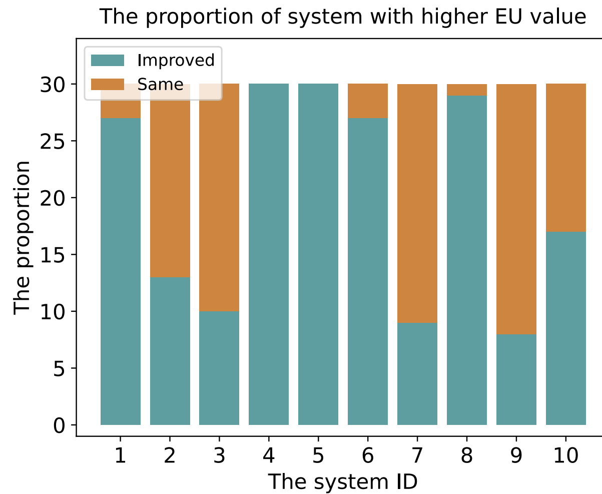

Figure 3.7: The proportion of systems with higher EU value under Util = 0.6 (10 randomized graph structures and 30 CPD tests for each bar)

Figure 3.7, 3.8, 3.9 and 3.10 illustrates the performance of systems with utilisation 0.6. Each system structure has 30 different CPTs. Figure 3.7 illuminates the proportion of the system with improved performance from an expected utility value (i.e., QoS) perspective, the blue part depicts the portion of the improved systems with higher QoS during graceful degradation. In contrast, the yellow portion illustrates the ratio of systems with the same performance. Figure 3.9 illuminates the proportion of the system with improved performance from a survivability perspective, and the grey part represents systems preserving more *LO*-criticality tasks, while the pink slice demonstrates the proportion of systems with the same level of survivability. One can observe that the improved proportions have almost the same value from both perspectives. That means the system with improved survivability would also

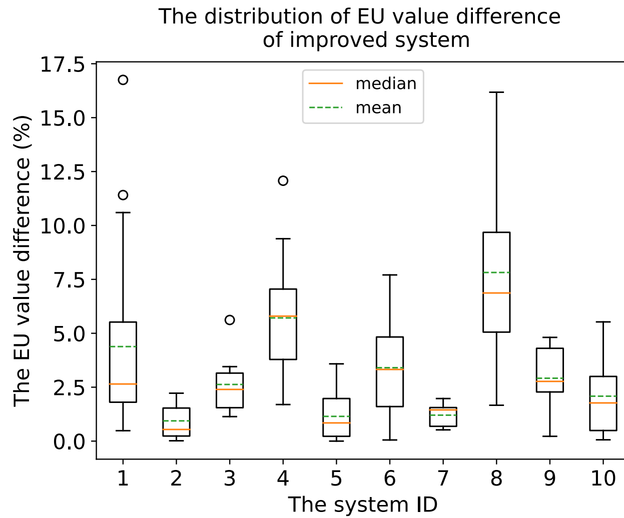

Figure 3.8: The distribution of EU value difference of improved system

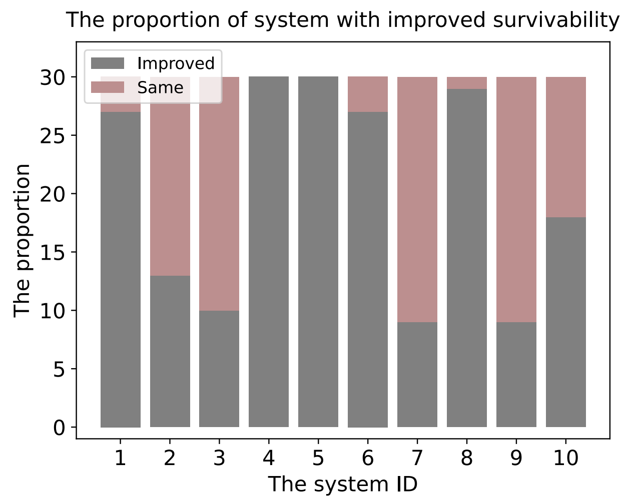

Figure 3.9: The proportion of systems with improved survivability under Util = 0.6 (10 randomized graph structures and 30 CPD tests for each bar)

maintain higher QoS during system overrun based on our proposed method. The histograms from the result of highly loaded systems with Uti = 0.9 also illustrate the same trend, as Figure 3.11 and 3.13 show; however, the overall proportion of improved systems significantly increased.

As the box plots of the EU value in Figure 3.8 and 3.12 show, based on our proposed method, the value of the improved graph can be maintained relatively higher. That implies that the system can be executed with a higher

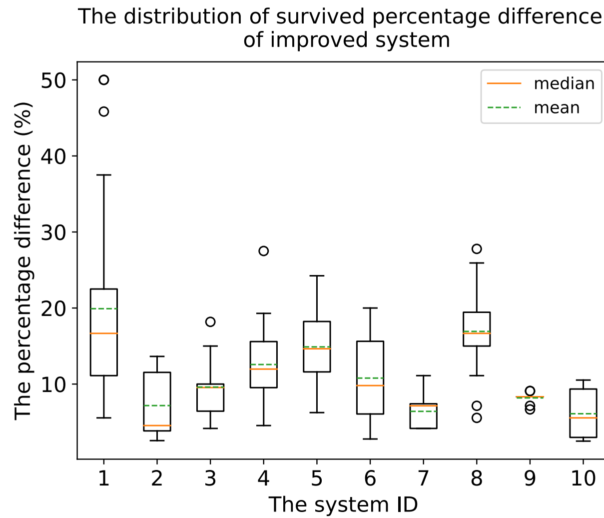

Figure 3.10: The distribution of survived percentage difference of improved system

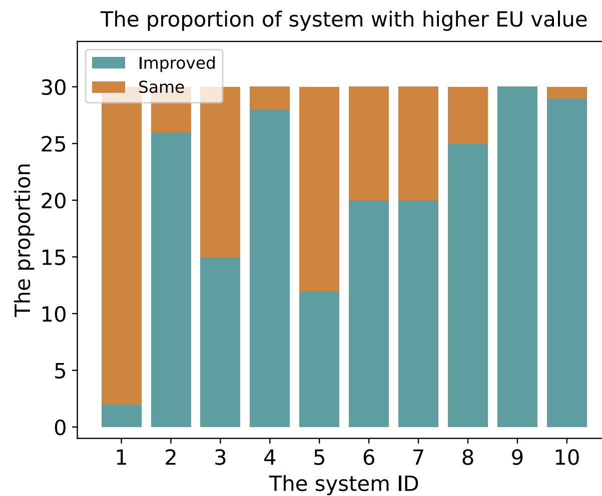

Figure 3.11: The proportion of systems with higher EU value under Util = 0.9 (10 randomized graph structures and 30 CPD tests for each bar)

QoS through graceful degradation. The green dashed lines denote the mean value, and the orange line represents the median value. Based on our proposed method, in the improved system with utilisation of 0.6 and 0.9, the EU value can be 3.8% and 6.2% more on average than the system without any QoS consideration, and the median value can be 2.6% and 4.9% higher, respectively. The magnitude of the difference in the EU value is directly af-

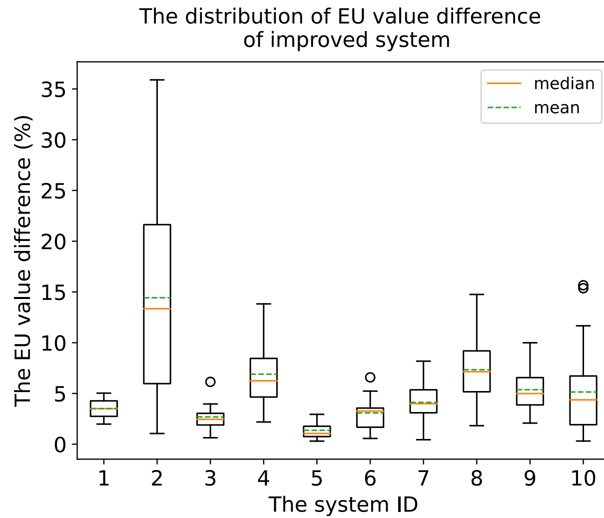

Figure 3.12: The distribution of EU value difference of improved system under Util = 0.9

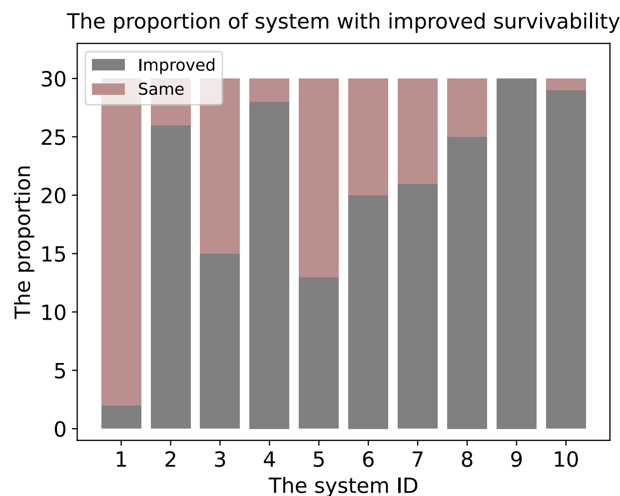

Figure 3.13: The proportion of systems with improved survivability under Util = 0.9 (10 randomized graph structures and 30 CPD tests for each bar)

fected by the value of the randomly generated probability value. If the value difference is quite small, such as $P_{correct} = 0.506$, $P_{incorrect} = 0.494$, and if the table values in the whole graph are similar, the variation of marginal value, which is used to determine the EU value, could be small. However, in the real world, the difference in probability value can be large enough to get a much more significant EU difference. Besides, a remedy in this situation is

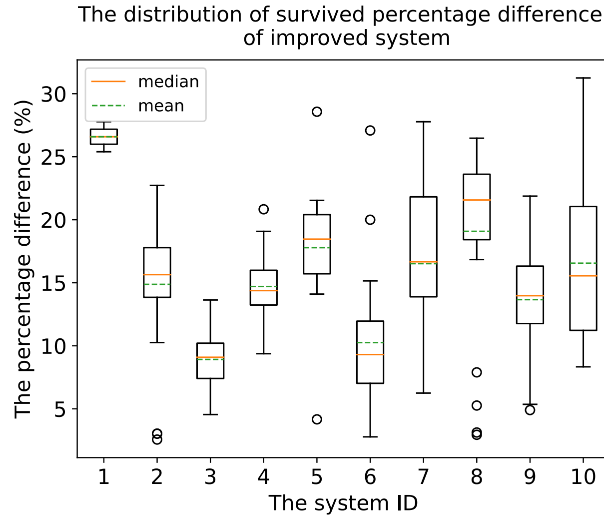

Figure 3.14: The distribution of survived percentage difference of improved system under Util = 0.9

to work with log messages, which is out of the scope of this work. The box plots showing survivability in Figures 3.10 and 3.14 illustrate that task-level fine-grained degradation control can enable more *LO* tasks to survive during system overrun. Under systems with utilisation 0.6 and 0.9, 12.7% and 15.0% more *LO* tasks on average can be saved, and the median value can be 11.5% and 15.0%, respectively.

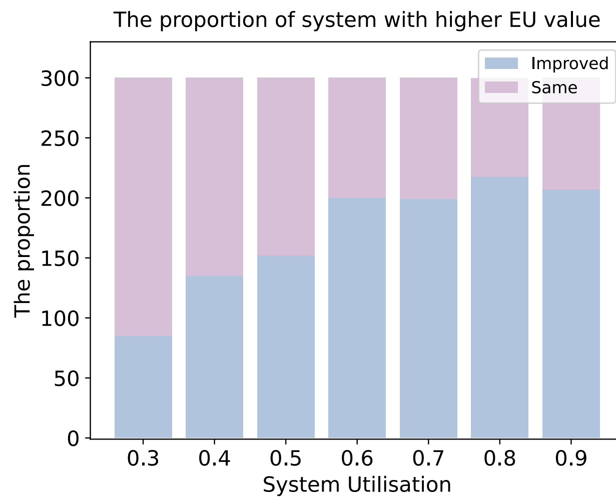

Figure 3.15: Experiment results for the proportion of systems with higher EU value under varied Util from 0.3 to 0.9 (each bar consists of 300 trials)

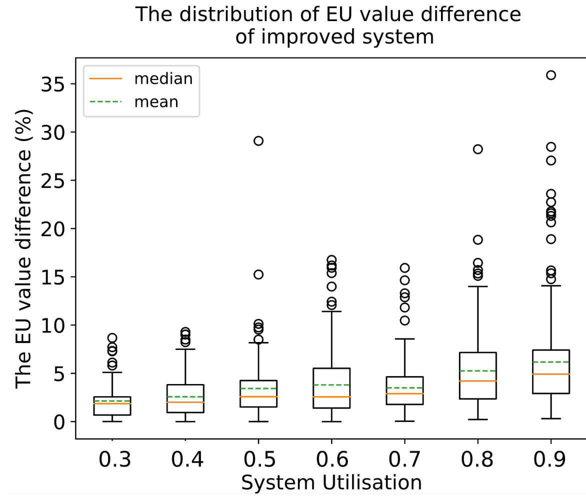

Figure 3.16: Experiment results for the distribution of EU value difference of improved system under varied Util from 0.3 to 0.9

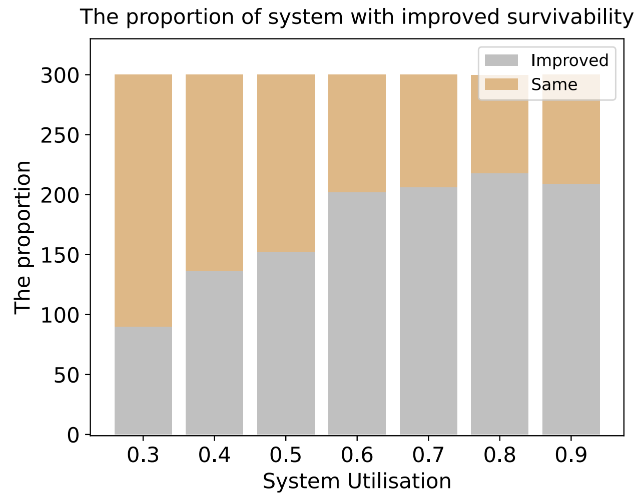

Figure 3.17: Experiment results for the proportion of systems with improved survivability under varied Util from 0.3 to 0.9

Figure 3.15, 3.16, 3.17 and 3.18 demonstrate the performance of our proposed method based on comprehensive results. The proportion of improved systems increases when we increase system workload, and the trends are consistent across the metrics of high EU value and survivability of *LO* tasks. The EU value can be maintained relatively higher during system overrun. As the median EU value difference in Figure 3.16 depicts, with the increase of the system load, the advantage of our proposed method with respect to maintain-

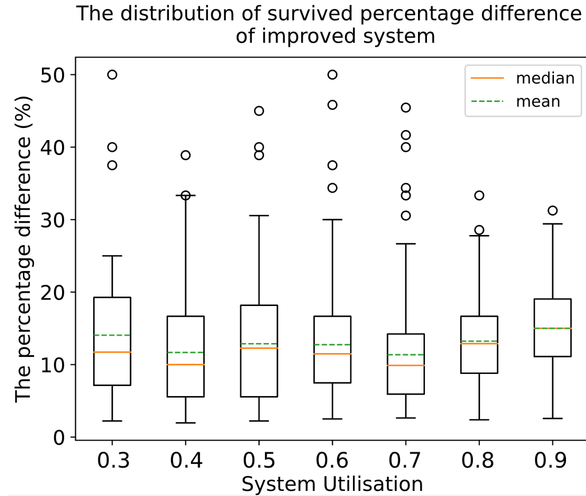

Figure 3.18: Experiment results for the distribution of survived percentage difference of improved system under varied Util from 0.3 to 0.9

ing QoS becomes significant. Furthermore, as the box plot about survived percentage in Figure 3.18 show, the task-level discarding method can better utilise the slack in the system because even in a highly loaded system, more *LO* criticality tasks can survive. However, we can observe that median value does not have a noticeable trend with the increase in system utilisation. That is caused by the strategy used for graceful degradation. Suppose dropping only one large *LO* task without considering QoS is enough at one task dropping point, which may lead to a significant decrease in system QoS. However, based on our proposed method, discarding the selected task with the lowest importance level may not be sufficient to release enough space for HI task execution. More tasks in the importance order should also be dropped to provide sufficient space. Although more tasks need to be dropped in some specific cases and similarly, task drop points, the reduction in system QoS may be slow, and the proposed method can still save more *LO* tasks throughout the degradation process.

3.6 Summary

In this chapter, we propose a context-aware, causality-based graceful degradation method with a new criterion by considering task dependency from the functional perspective, to assist in achieving more fine-grained task-level

degradation under different scenarios in an MCS setup. The survivability of *LO* criticality tasks in the system can be improved, thus giving more graceful degradation. At the same time, the awareness of system-wide influence propagation can ensure that the discarding decision is capable of minimising the reduction of the systems' QoS. Furthermore, considering belief value variation under different circumstances provides more flexibility to the fixed degradation order. Compared with the standard application-level discarding method, our proposed method enables over 10% more *LO* tasks to survive in a highly loaded system. The expected utility value can be maintained around 5% higher.

Chapter 4

Resilience-aware Multi-core Mixed-criticality Consistent DAG Scheduling

As described in Chapter 1, advanced driver-assistance and semi-autonomous systems are complex and safety-critical with strict real-time and resource constraints, as well as having a deep processing pipeline with strong dependencies across functions. In such systems, tasks with different criticality levels are integrated on the same hardware platform, thus forming a mixed-criticality system. The scheduling strategy has to guarantee that higher criticality tasks have no risks introduced by lower criticality tasks. To do so, most static scheduling work considers a dual-criticality system and assumes that all *LO*-criticality tasks can be suspended or discarded after a mode change. However, this makes the schedules of HI and LO modes different, requiring more effort to verify the safety of schedules both during and after mode changes and potentially raising migration costs, which is even more complicated with the extension to multiple criticality levels. Moreover, though slack management can allow the execution of tasks with lower criticality in relatively higher modes, it is still a challenge for existing methods to guarantee the effectiveness of slack utilisation and precedence constraints simultaneously. This chapter proposes a novel, multi-core mixed-criticality consistent DAG static scheduling (namely *mccs-dag*) method with limited preemption, which avoids migration. Instead of conventional static schedules for MCS (i.e., different schedules for different system modes), tasks with different criticality levels are maintained

based on a single schedule. Furthermore, task level mode change improves the survivability of tasks with lower criticality value.

4.1 Introduction

In recent decades, the development of computational hardware has facilitated the implementation of more advanced algorithms, enabling autonomous systems to undertake complex tasks. However, advanced computational units such as GPUs and FPGAs are expensive and size-constrained, limiting the hardware resources for commercial systems. This raises difficulties in functional integration, as tasks in autonomous systems often have a high degree of dependency, with some tasks only starting to run when all their predecessors have completed their execution. Furthermore, tasks have different levels of criticality defined according to safety standards such as ISO 26262 [2] and DO-178B [3], which further increase the complexity of scheduling design, especially to guarantee the execution of tasks with certain critical levels in the corresponding system modes and with hard real-time requirements.

To address these challenges, multi-core scheduling for mixed-criticality systems (MCS) has become an emergent trend in the real-time scheduling community. MCS integrate multiple functions of different criticality on a shared platform, where the parallelism realised by multi-cores can significantly benefit computation-intensive workloads. However, most work has assumed that tasks are independent and adopted the classic MCS model introduced by Vestal [120] in which, for dual criticality systems, all tasks are initially executed in the LO criticality mode. Once a timing fault such as overrun happens in any HI criticality task, the system mode changes from LO to HI, after which all LO criticality tasks are suspended or discarded. Only HI-criticality tasks are allowed to run to their maximum estimated worst-case execution time (WCET). As emphasized in Chapter 3, Bletsas et al. observed that although some tasks might be considered as LO criticality, in practice, they might still contain mission-critical functions and be vital for the correct and efficient operation of the system [26]. Consequently, Burns et al. further emphasized the importance of the robustness and resilience of MCS task scheduling [32]. Furthermore, it has been shown that the overrun of one specific HI criticality task does not imply that all HI tasks simultaneously run up to their largest WCETs [28]. Therefore, it is possible and desirable to provide the system designer with more

fine-grained control over the system degradation and improve the survivability of lower criticality tasks, such as task-level mode change involved in our work.

As well as improving the system's resilience, the fundamental scheduling strategy is vital to ensure the execution of tasks. The flat-structured scheduling method, discussed in Chapter 3, provides an opportunity to realise task-level degradation. However, this still results in unnecessary task discarding. Fohler and Baruah noted that static scheduling is completely deterministic and well-supported in the industry, as it is easy to certify [21]. For safety-critical systems, the scheduling strategy must be predictable and certifiable. Therefore, we adopt a static scheduling method and use Directed Acyclic Graphs (DAGs) to model task precedence constraints. However, DAG scheduling on multiprocessors is challenging, and the problem becomes more complex for mixed-criticality systems [81]. Extending a timing budget may have a cascade effect on dependent tasks. Therefore, scheduling design should take special care to ensure the satisfaction of precedence constraints and schedulability.

In this work, we aim to address the task scheduling problem for the entire system by scheduling multi-periodic Directed Acyclic Graphs (DAGs) of mixed-criticality tasks on a multi-core platform. We propose a static scheduling method based on a more practical system model. Our method considers that tasks with higher criticality levels may depend on tasks with lower criticality levels, which breaks the constraints used by most MCS static scheduling methods. In Section 4.2, we will use a real application example from autonomous systems to explain the necessity of breaking this constraint. This means that the parents of a safety-critical task can be of low criticality, but at least one task from its parent task group should be safety-critical to guarantee the fundamental safety operation. Compared with the state-of-the-art static scheduling method $lsai - edf$ proposed in the paper [93], the proposed method employs a unified scheduling approach to manage the execution of tasks with diverse criticality requirements and facilitates task-level mode changes, which are not efficiently attainable using the $lsai - edf$ method. The higher LO task survival rate enhances the system's resilience and optimizes resource utilization, while the increased degradation rate can further expedite system recovery.

Our **main contributions** can be summarized as follows:

- To the best of our knowledge, this is the first work to propose a static schedule-based task-level mode change for mixed-criticality systems with

multiple criticality system modes so that expensive system-level mode changes can be eliminated and more tasks under different system modes can be preserved. The safe execution of tasks under different system modes is guaranteed based on our proposed start time recalculation method during static schedule calculation.

- Instead of maintaining a mixed-criticality system with different schedules, this is the first work to use a single static schedule to maintain the execution of all tasks with different criticality levels. The safe execution of higher-criticality tasks can be guaranteed in any system circumstance.

- As a general method, the proposed mixed-schedule can generate a safe and consistent schedule that enables task-level mode change and can be easily generalized to multi-criticality systems. Verification and calculation based on overrun testing can provide more opportunities to save lower-criticality tasks when overruns occur, thus improving the system's resilience and quality of service.

The rest of this chapter is organised as follows: In Sec. 4.2, the proposed scheduling method is detailed, and this is followed by the formulation of the proposed consistent scheduling strategies in Sec. 4.3. The evaluation is introduced in Sec. 4.4. Finally, in Sec. 4.5, we summarise this chapter.

4.2 Method Overview

In this work, we propose one comprehensive general mixed-criticality scheduling procedure. As illustrated in Figure 4.1, the procedure can be divided into three layers. At the top layer, the effort would focus on functional dependency analysis and segmentation, followed by the middle layer task group to core cluster mapping. The workload (utilisation) of different core clusters should be balanced at this layer. Please note that the middle layer of Figure 4.1 shows a simple hardware structure example, and in practice, the processing resources could consist of several core clusters and in each cluster, there could be more than four cores. Finally, the most critical task scheduling strategy would be carefully designed at the bottom layer. The main contribution of this work is from the bottom layer resilience-aware consistent mixed-criticality DAG scheduling. The methods at the top and middle layers will be briefly

introduced using simplified examples. We assume that each function can be modelled as mixed-criticality DAGs, and functions can have different periods.

Most existing mixed-criticality scheduling methods are recognized as flat-structured scheduling methods, where a central scheduler is used to schedule all tasks in the entire system. However, some complex systems, such as autonomous systems, consist of a set of functions with dependencies, and there are also some functions that can work independently. For example, mission planning is designed based on object detection, localization, and mapping, and the hardware diagnosis module may not have any relationship with mission planning and can work independently. Furthermore, each function would consist of tasks with dependencies and different criticalities. Taking the object detection function of the autonomous system as an example, it may consist of three redundant elements — camera-based, LiDAR-based, and radar-based functions. This function is highly safety-critical, but solution designers can decompose the criticality across the redundant elements and meet safety goals [61]. Suppose the LiDAR- and radar-based sub-functions are classified into the HI criticality task group by the system designer. The camera-based sub-function can be discarded when the system cannot tolerate the overrun of HI criticality tasks.

4.2.1 Top Layer: Functional Dependency Analysis and Segmentation

In this subsection, we briefly introduce the method for the top layer. Figure 4.2 illustrates one simplified dual-criticality system example, which consists of 5 functions (A, B, C, D, and E) and each function is composed of a different number of tasks with precedence constraints. All tasks in the system can be categorized into two criticality levels, HI-criticality and LO-criticality, which are marked by orange and blue colours, respectively. Obviously, functional dependency exists between A and C, B and C, and B and E. Function D is independent.

We can use joint distributions to describe functional dependencies. The graph-segmentation algorithm, such as D-segmentation [80], can be used to segment the system graph into the expected number of sub-graphs and determine the dependency level of each function. In this example, assuming we have two core clusters, each consisting of four cores, the system graph should be segmented into two groups. The adoption of graph segmentation aims to

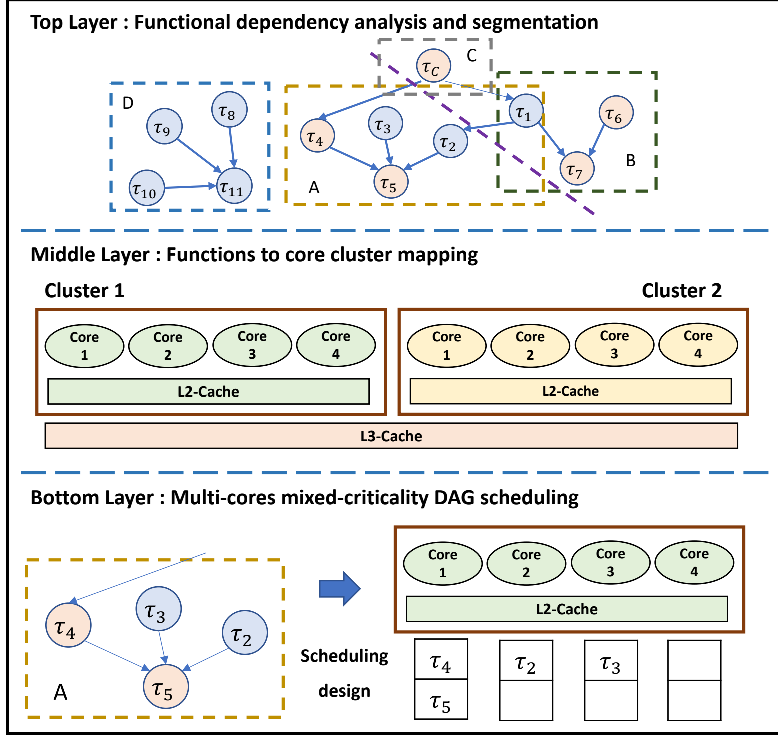

Figure 4.1: The proposed system-level scheduling architecture

reduce the data transmission (failure propagation) between two core clusters. Moreover, considering the highly efficient utilisation of shared resources, such as cache, functions with relatively tighter dependency are better allocated to the same core cluster. Thus, in this example, function D will initially be mapped to group one, and functions A, B, C, and E will be clustered into group two.

4.2.2 Middle Layer: Functions to Core Cluster Mapping

If we directly use the task groups obtained from the top layer, we may face severe load imbalance, i.e., there is only one function in group one and four functions in group two. Thus, some functions should be migrated from the high-load cluster to the light-load group. We will start from the function with the lowest dependency level, which is determined by the number of functions that have relationships with it (i.e., the number of predecessors and successors). Among the functions with the same dependency level, the one that can primarily ease the imbalance difference would be selected first. In this example, functions A and E have the same dependency level with the number

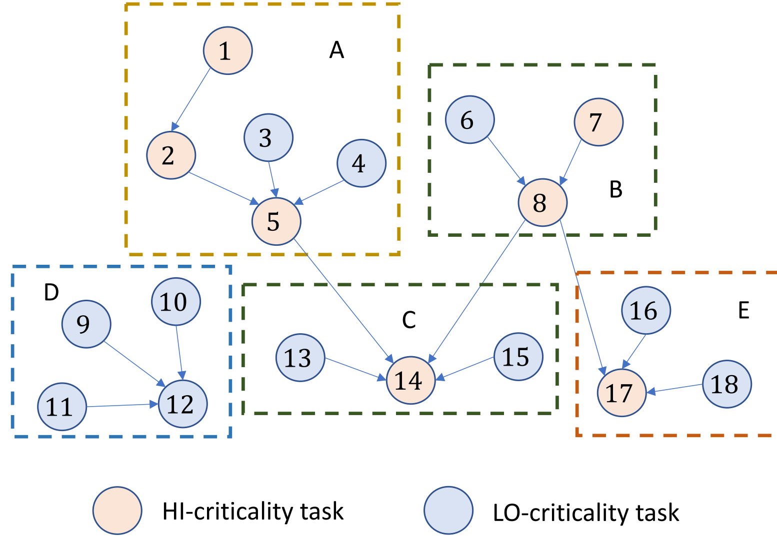

Figure 4.2: Simplified example of a dual-criticality system

of predecessors and successors (i.e., one). If the migration of function E can minimize the utilisation difference between the two task groups, all tasks in function E will be moved to group one and scheduled with function D.

4.2.3 Bottom Layer: Multi-core Mixed-criticality DAG Scheduling for System with Multiple Criticality Levels

As the most important contribution of this chapter, this section introduces the task model and the proposed resilience-aware mixed-criticality consistent static DAG scheduling method. We adopt a dual-criticality system for a motivating example, which intuitively and understandably demonstrate the shortcomings of existing methods and the benefits brought by our proposed strategy. However, our method is generalisable to systems with more than two criticality levels.

Mixed-Criticality System Model:

We adopt criticality-dependent WCET estimations. A task τ_i can be defined by a tuple $\tau_i := (T_i, D_i, C_i^{L_i}, L_i)$, where T_i and D_i represent the period and deadline, respectively ($D_i = T_i$). L_i is the task criticality level. For an M-criticality system, the system can run in M different criticality modes $S = \{1, \ldots, M-1, M\}$. $S_1 = 1$ represents the lowest system criticality mode and is regarded as a normal mode, in which all tasks can be executed ide-

ally with their $C(S_1)$. In each system criticality mode, the tasks that must be guaranteed are predefined according to the task's criticality level. Take a dual-criticality system (i.e., $S = \{LO, HI\}$) as an example: in HI system mode, all tasks with HI-criticality level (i.e., $L_i = HI$) should satisfy their timing constraints. For each task, $C_i^{L_i}$ denotes the set of execution times under different system criticality modes (i.e., $C_i^{L_i} = C_i^{L_i}(S_1), \ldots, C_i^{L_i}(S_{M-1}), C_i^{L_i}(S_M)$). Conventionally, for a dual-criticality system in HI mode, all tasks with HI-criticality level must be guaranteed with their estimated WCETs in HI mode (i.e., $C_i^{HI}(HI)$), and all LO-criticality tasks are suspended or discarded. Thus, we don't have WCET estimation for LO-criticality tasks in HI mode. In LO mode, both HI and LO-criticality tasks should finish their execution before deadlines with estimated WCETs in LO mode (i.e., $C_i^{L_i}(LO)$). Therefore, for each HI-criticality task, there are two different WCET estimations, i.e., $C_i^{HI}(LO)$ and $C_i^{HI}(HI)$, with $C_i^{L_i}(LO) < C_i^{L_i}(HI)$. When extending to multi-criticality systems, $C_i^{L_i}(S_1) < \cdots < C_i^{L_i}(S_{M-1}) < C_i^{L_i}(S_M)$, where $L_i \geq S_M$. There is no WCET estimation if $L_i < S_M$.

In this work, lower-criticality tasks should survive as long as possible. If not possible, they will always try to keep a simple data refresh operation before being discarded to avoid interfering with the execution of higher-criticality tasks. Data refresh is seen as a minimum operation and recognized as the degraded operation of a droppable task to ensure the timeliness of data. For example, when the LO-criticality camera-based lane detection task transitions to the degraded state, it only needs to update the received image data without additional processing and simply publish a flag message to its successors. The time taken by the degraded task is significantly shorter than the estimated WCET $C(LO)$. If the task can be recovered, the time consumed to recover from the degraded state is also much faster than from the discarded state. The experiment presented in Section 4.4 provides evidence for this claim. We use δ_i to represent the minimum operation of a droppable task. Thus, for a task τ_i with criticality level N (i.e., $L_i = N$) in S_M system criticality mode, $C_i^N(S_M) = \delta_i$, if $S_M > N$. For example, in a dual-criticality system, $C_i^{LO}(HI) = \delta_i$. Besides, we assume that in each specific criticality system mode (e.g., the system is running in M-criticality system modes), all tasks with criticality higher or equal to the system mode (i.e., $L_i \geq S_M$) must be guaranteed, while tasks with criticality lower than S_M can be degraded with $C_i^N(S_M) = \delta_i$ or discarded with $C_i^N(S_M) = 0$.

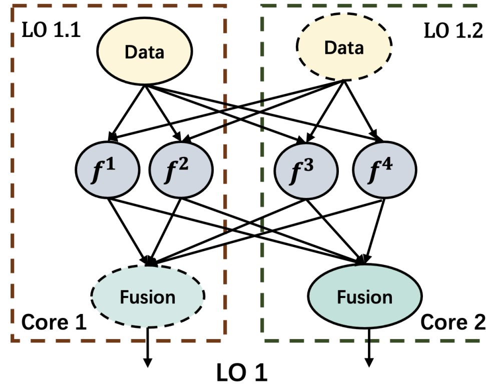

Figure 4.3: Example of LO-criticality task segmentation in a dual-criticality system

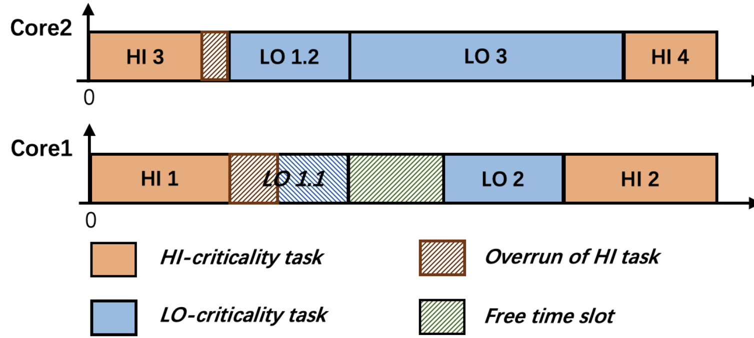

Figure 4.4: Example of HI-mode schedule with degraded LO-criticality task in dual-criticality system

Additionally, system designers may adopt the multi-input single-output concept to increase the confidence measures in deployed algorithms. Multiple frames would be cached and fed into the same type of function to enhance the confidence in the output by fusing multiple outputs. To accelerate the computation, the function can be executed in parallel. The absence of an output from a single function may only imply performance degradation, and the functionality can be regarded as having survived. Figure 4.3 demonstrates an example.

Example: Figure 4.4 illustrates an example schedule including the kinds of tasks described in Figure 4.3. LO-criticality function LO_1 can be segmented into two sub-tasks (i.e., $LO_{1.1}$, $LO_{1.2}$) and deployed to two different cores. Task LO_3 cannot start its execution before receiving outputs from $LO_{1.1}$ and $LO_{1.2}$.

As Figure 4.4 shows, the overrun of HI_1 task only impacts $LO_{1.1}$. According to the strategy adopted in our method, $LO_{1.1}$ will maintain a minimum data refresh operation, recognised as a degraded task, because after HI_1 finishes its execution with $C(HI)$, the free time slot before LO_3 is sufficient to support the minimum operation of $LO_{1.1}$. Additionally, the overrun of HI_3 is not severe enough to impact the execution of $LO_{1.2}$. Therefore, $LO_{1.2}$ can start its execution as expected and be recognised as survived. Based on this, task LO_1 is considered as surviving with degraded performance. This example highlights the importance of preserving all LO criticality tasks as long as possible, because at runtime, it is impossible for the execution schedule to differentiate tasks' functionality. Moreover, it is worth noting that if the end time of HI_3 and HI_1 never exceeds the predefined start time of LO_2 and LO_3, even with C(HI), then LO_2 and LO_3 will never be discarded, as their execution will never be affected. An appropriate static schedule can facilitate the task-level mode change because, for each HI task, the LO tasks that could be impacted by its overrun can easily be identified once the schedule is known.

System structure model:

Based on an understanding of task dependencies in autonomous systems, we employ directed acyclic graphs (DAGs) to depict these task relationships. In this study, we consider a system S comprising N applications, denoted as $S = \{A_0, A_1, ..., A_N\}$. Each application is composed of tasks (nodes) with dependencies ($G_{A_x} = \tau_0, \tau_1, ..., \tau_m$), represented by relatively small DAGs. The interconnections among these application DAGs collectively form a larger system-level DAG G_S.

System mode change v.s. Task-level mode change

In mixed-criticality systems, a system mode change refers to the transition between different operational states. This thesis specifically means the transition between different criticality system modes. In multi-criticality systems, if the actual execution time of any task with a criticality value higher than S_M ($L \geq S_M$) exceeds $C(S_{M-1})$, the system needs to change to the M-criticality mode. In this mode, all tasks with criticality levels equal to or higher than S_M are allowed to run up to their maximum estimated WCET $C(S_M)$, while tasks with criticality values lower than S_M need to be suspended or dropped. For example, in a dual-criticality system, if the actual execution time of any

HI-criticality task attempts to exceed $C(LO)$, then the system should transition to the HI-criticality mode. In this mode, all LO-criticality tasks should be suspended or dropped.

To enhance the survivability of droppable tasks in various system modes, some research works, such as those reviewed in Section 2.7, aim to employ free budget allocation methods to maintain more fully executed droppable tasks or reduce QoS to prevent a complete loss of functionality. However, none of these approaches is based on a static schedule with task predecessor constraints. This is because budget allocation cannot assign a specific time slot to a dropped task to ensure its execution after the completion of its predecessors and before the initiation of its successors. This is the first work aimed at proposing a static schedule calculation method that considers task dependency and survivability simultaneously.

As previously introduced, for static schedule-based MCS, we require different schedules to manage task execution. Figure 4.5 illustrates example static schedules for a triple-criticality system proposed in paper [93], where χ_1 represents the lowest criticality system mode. Tasks from the same application are marked with the same color, and this example illustrates the static schedule of a system consisting of two applications. It is evident that the schedules differ. In each criticality system mode, only tasks with higher or equal criticality levels are scheduled using their corresponding WCET. Based on such schedules, the only way to guarantee the safety of tasks is through a system mode change. If tasks with higher criticality do not overrun to their worst case, we can examine the free slot for the dropped tasks between their predecessors and successors to find an opportunity for their execution. For different schedules in different system modes, this is not easy, especially when all tasks are executed as late as possible. Although free time slots exist at the beginning of one hyper period, they cannot be used due to task precedence constraints. Another disadvantage of the method proposed in paper [93] is the extremely high preemption rate and migration rate. Therefore, the authors only pointed them out and clarified that their evaluation did not consider the overhead brought by preemption and migration; however, in practice, this will significantly reduce schedulability. For example, based on their method, tasks can be preempted every single time unit. Thus, the actual schedulability rate should be lower than their evaluation results. To provide a more practical solution, we introduce a limited preemption method and avoid migration at

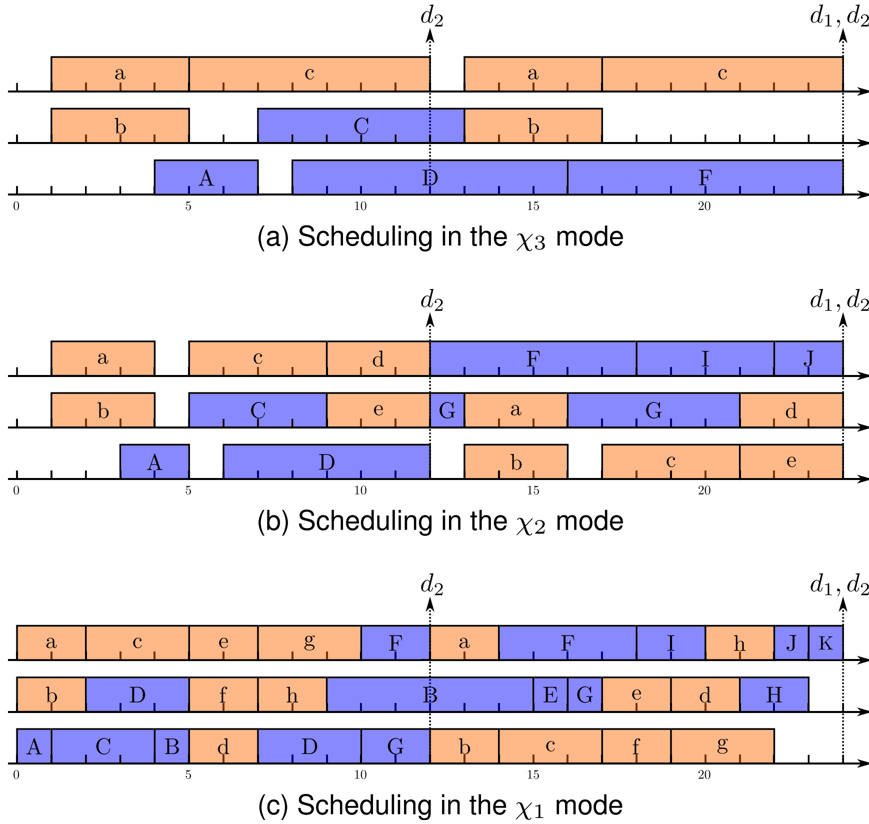

Figure 4.5: Static schedules example for multi-criticality system [93]

runtime to considerably reduce the overhead.

In this thesis, task-level mode change can be described as follows: even when the system has transitioned to a higher criticality system mode, tasks with a criticality level lower than the system criticality level can continue their full execution. Only the impacted droppable tasks need to be degraded or discarded. For example, in a triple-criticality system (i.e., LO, MI, HI), initially, all tasks operate with their $C(LO)$. If any MI or HI-criticality task attempts to exceed $C(LO)$, the system could transition to the MI-criticality mode. Then, all MI and HI-criticality tasks are allowed to run up to their maximum estimated WCET $C(MI)$. The LO-criticality task impacted by the specific overrun MI or HI tasks can be degraded or discarded, while all other unaffected tasks can be executed as normal. When any HI-criticality task attempts to exceed $C(MI)$, the system starts working under HI-criticality mode. All MI and LO-criticality tasks become droppable, and the impacted ones can be degraded or discarded. This is exemplified by the motivational

example for a dual-criticality system in Figure 4.4. To further ensure that tasks from lower criticality levels do not impact tasks with higher criticality levels, priorities can be assigned such that tasks from higher criticality levels receive a higher priority value than tasks from lower criticality levels within each core. Based on this, droppable tasks from higher criticality levels can preempt the execution of lower-criticality tasks.

Based on task-level mode change, more droppable tasks can be preserved in any criticality system mode. To achieve this, a consistent schedule is required. This means that we need to use a single schedule to maintain task execution in any system mode, as the impacted tasks analysis should be based on a fixed schedule that cannot be changed during system mode transitions. In the rest of this section, the methods for calculating a consistent schedule will be detailed and described.

Consistent Mixed-Criticality Static DAGs Scheduling:

We hereby propose a new *Consistent Mixed-Criticality DAGs static scheduling* (*mccs-dag*) method. The main procedure is illustrated by Algorithm 5. Unlike state-of-the-art static scheduling methods, we relax the criticality dependency constraints, i.e., a higher criticality task is allowed to depend on lower criticality tasks, but at least one predecessor must have the same criticality level as the analysed task. The relaxed constraints make the DAG model more practical. When facing an overrun of a specific higher criticality task, a task-level mode change is adopted, i.e., only affected lower criticality tasks would be degraded or discarded rather than simply discarding all lower criticality tasks simultaneously. Once the execution of the higher criticality task is restored, the impacted task can immediately return to normal mode without the need for costly system restoration. This is one of the claimed benefits of the task-level mode change strategy. If the task returns from a degraded state, the recovery time can be even shorter, which can be proved by the real experiment described in Chapter 4.4.

For DAG scheduling, the execution of one task can only be started after all its predecessors are finished. When calculating schedules with backward propagation, a task can only be treated after the time slots for all its successors have been allocated. Therefore, the definition of each task should be extended to $\tau_i := (T_i, D_i, C_i^{L_i}, L_i, Suc(\tau_i), Pred(\tau_i))$, where $Suc(\tau_i)$ and $Pred(\tau_i)$ represent the set of successors and predecessors of τ_i, respectively. Please note that

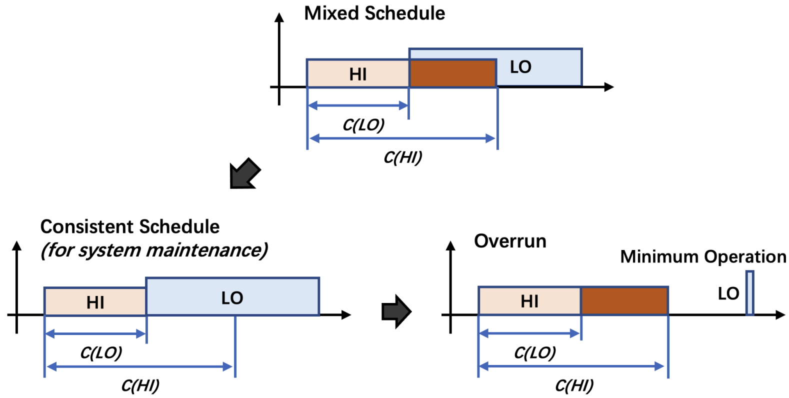

Figure 4.6: An illustrative example of a mixed schedule

tasks from the same DAG have the same period and deadline.

Example: Instead of calculating schedules for different system criticality modes separately, in this work, we introduce a mixed-schedule that provides a safe schedule and enables task-level mode change. For a dual-criticality system, as illustrated in Figure 4.6, the mixed schedule has an overlapping region between HI and LO tasks, which equals the execution time difference of HI-criticality tasks under different system modes. Shrinking the HI tasks can help us obtain a consistent schedule that guarantees the execution of tasks in low-criticality mode.

Based on the consistent schedule, once an overrun of any HI task happens, the impacted LO task can be easily identified. In the example of Figure 4.6, there is sufficient slack time to support its minimum operation during overrun. Importantly, this consistent schedule enables task-level mode change. The start time of HI tasks remains the same in any system mode. Once the execution of any HI task returns to normal mode, the impacted LO tasks can be directly restarted without any effort and overhead raised by conventional system mode change. Furthermore, with the elimination of a system mode change operation, we do not need to check the schedule's safety during mode change.

Figure 4.7 illustrates the schedule allocation procedure for a mixed schedule. Like others, e.g., [26], we schedule tasks in a backward direction (i.e., starting from the sink node of the DAG towards the source node). The task is selected according to its priority from a waiting queue, and Sec. 4.3 will detail the method for waiting queue generation and priority allocation. Suppose the selected task is LO-criticality; it will be allocated with $C(LO)$ after passing

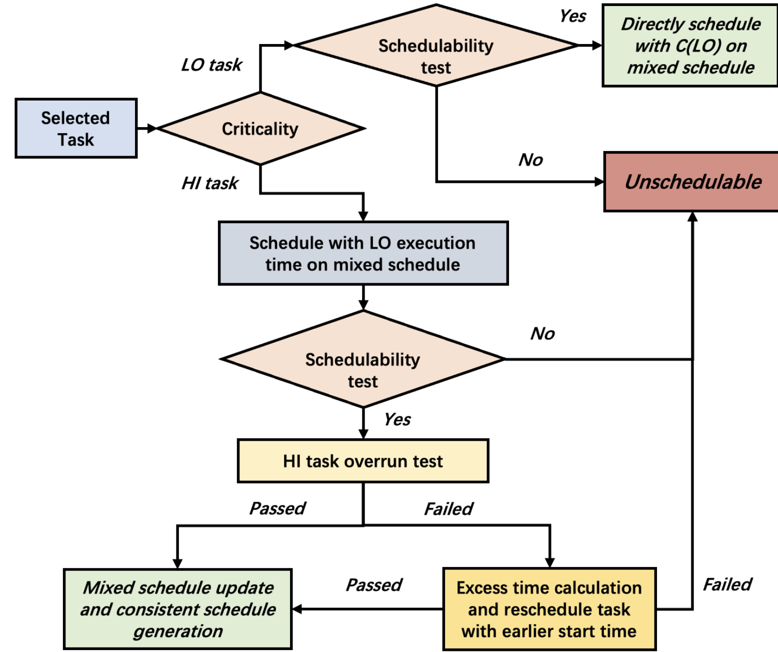

Figure 4.7: Selected task scheduling flowchart

the slack time analysis-based schedulability test, which requires the sum of valid slack time intervals to be larger than $C(LO)$ (line 9 - 10 of Algorithm 8). Otherwise, the system is deemed unschedulable. To limit the preemption overhead, we adopt a limited preemption strategy. The minimum execution fragment should be larger than a specific value, which can be determined according to the execution time estimates or using empirical values. Therefore, an individual slack time interval will be invalidated if it is shorter than the minimum execution fragment length (Algorithm 9). The time interval for slack time analysis can be constrained between the earliest allowed start time of the selected task (i.e., the earliest release time of the DAG to which the selected task belongs) and the earliest start time of its successors. For HI-criticality tasks, in addition to the similar slack analysis as for LO tasks, the execution time slots should be further calculated based on the overlapping region analysis during the overrun test.

Algorithm 5: Consistent Mixed-Criticality DAGs Static Scheduling

```
   /* Initialise global system parameters                         */
 1 Initialisation();
 2 WaitingQueue ← UpdateWaitingQueue();
   /* Try to schedule each task in the waiting queue              */
 3 while WaitingQueue ≠ ∅ do
 4     τ ← pop(WaitingQueue);
 5     Dry run task schedule on each core;
 6     ζ_τ^m ← ScheduleOnCore(t, m, true) ∀m ∈ 1 : N;
 7     ι ← argmax_m(ζ_τ^m); /* Select the core to schedule        */
 8     if ζ_τ^ι = ∞ then
 9         return Unschedulable; /* Check schedulability          */
10     end
11     else
12         ScheduleOnCore(t, ι, false); /* Schedule update        */
13     end
14     WaitingQueue ← UpdateWaitingQueue();
15 end
16 return ℰ /* consistent schedule generation                      */
```

PseudoCode

Algorithm 5 illustrates the main pipeline of the proposed method. Unscheduled tasks from each DAG are fed into the *Waiting Queue* (i.e., Algorithm 6) when all of their predecessors have been scheduled and removed. The task with the highest priority from the *Waiting Queue* is selected and scheduled according to the proposed schedule calculation method (i.e., Algorithm 7 and Algorithm 8). After allocating an appropriate core and time for the selected task, the system's parameters, including the *Waiting Queue*, utilization of each core, and mixed schedule, are updated before starting the next task scheduling calculation round. The loop will continue until all tasks in the *Waiting Queue* are scheduled, or an unschedulable task is encountered.

It is worth noting that our proposed method does not allow tasks to migrate to another core during schedule calculation, which will be fixed at runtime. Therefore, it does not incur any migration costs. This makes it easy to allocate higher criticality tasks with higher priority on each core, which can further

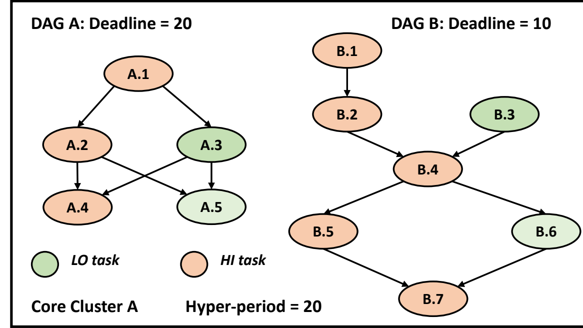

Figure 4.8: Mixed-criticality DAGs example from a dual-criticality system

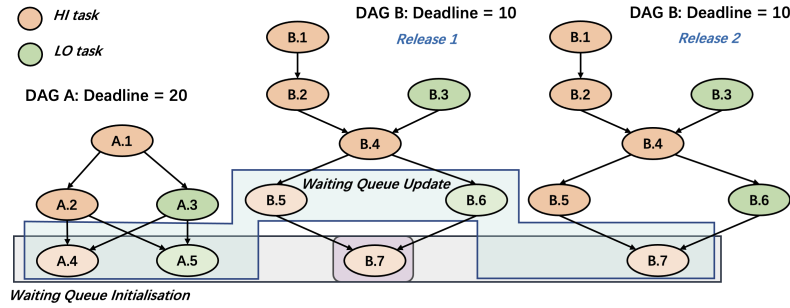

Figure 4.9: Initialisation and update of waiting queue

provide a preference for tasks with higher criticality levels.

4.3 Consistent Schedule Formulation

In this section, the proposed strategy is formulated and detailed in four steps: (a) waiting queue initialisation and dynamic updating, (b) priority definition and task selection, (c) mixed schedule-based consistent schedule calculation, and (d) parameters and consistent schedule update.

4.3.1 Waiting Queue Generation

To provide the most flexibility for schedule allocation, we adopt the idea proposed in [91]: the schedule calculation starts from the last time point, i.e., the last task of the DAG back to the source node. This approach allows all tasks to be executed as late as possible. In DAG scheduling, tasks can only be scheduled when all their successors have been scheduled. In our work, a system consists of more than one DAG with different periods, and each DAG could be released several times within their hyper-period. Figure 4.8 depicts

```
1  Function Initialisation():
2      Initialise all tasks for all DAGs and make available for all
         functions;
3      Γ; /* All task IDs, Γ represents a task set that
           includes all tasks τ in the system.                     */
4      for τ in Γ do
5          T_τ; /* Task period                                     */
6          D_τ; /* Task deadline, in this work D_τ = T_τ          */
7          L_τ; /* Task criticality level                          */
8          S; /* System criticality mode                           */
9          C_τ^{L_τ}(S); /* Task execution time in S criticality
               system mode                                         */
10         P_τ; /* Task priority                                   */
11         SD_τ ← false; /* SD represents the state of each
               task and initialised as not scheduled               */
12         Suc(τ); /* Successors                                   */
13         Pred(τ); /* Predecessors                                */
14         ζ_τ ← ∞; /* Start time                                  */
15         T_st^τ; /* Period start time of τ                       */
16         P_ed^τ; /* Period end time τ                            */
17     end
18     Initialise schedule and Mixed schedule ℰ for all cores;
19     ι is the core ID and N represents the number of cores;
20     for ι ← 1 : N do
21         ts represents each time stamp in the hyper period;
22         for ts ← 1 : hyperperiod do
23             ℰ_ts^ι ← ∅;
24             ℰ_ts^ι(S) ← ∅; /* Mixed schedule on system mode S   */
25         end
26     end
27     Construct waiting queue;
28     WaitingQueue ← ∅;
```

a dual-criticality system consisting of two functions deployed on one specific core cluster A. The period of functions is 10 and 20 time units, respectively. As shown in Figure 4.9, the hyper-period of the system is 20. Thus, each release of DAG A and DAG B will be regarded as an independent DAG in the system with a different release time.

For the convenience of task selection, we use a waiting queue for all ready tasks, and the definition of ready is only based on the satisfaction of task dependency. To start with, the sink node of each DAG will be fed into the waiting queue. In this example, $A.4$, $A.5$, and the first and second released $B.7$ are enqueued. (Please note that within a single hyper period, a DAG could be released multiple times. Each instance is recognized as an independent DAG with a distinct earliest start time. In such cases, the sink nodes from all duplicated DAGs can be simultaneously fed into the waiting queue, and which one from them will be selected first is based on the calculated priority introduced in the following subsection.) Then, one task will be selected according to its priority. If the first released task $B.7$ is selected and scheduled, its predecessors $B.5$ and $B.6$ can be fed into the waiting queue because all their successors have been scheduled. Suppose the currently selected task is $A.4$, $A.2$ cannot be enqueued before the schedule of $A.5$ is completed.

Algorithm 6 illustrates pseudocode for the generalized waiting queue update process.

4.3.2 Priority Assignment and Task Selection

As mentioned in the previous section, tasks in the waiting queue will be selected based on their priority. Any priority assignment policy can be employed according to the proposed architecture. This work uses a widely used critical path and deadline-based priority assignment policy similar to the one in paper [85].

The critical path of a vertex τ_i is the longest path, considering the execution time of vertices in the highest system mode (e.g., $C_i^{L_i}(HI)$ in dual criticality systems), to reach an exit vertex (i.e., a vertex with no successors). Suppose each DAG can be scheduled independently, and the finishing time $f(\tau_i)$ of each node can be calculated using Equation (4.1), where $CP(\tau_i)$ denotes the critical path of task τ_i; G_i represents DAG i; and N_{G_i} represents how many times the DAG is released within the hyper-period.

Algorithm 6: Waiting Queue Update

```
1  Function UpdateWaitingQueue():
2      for τ in Γ do
3          if SD_τ = false then
               /* τ is unscheduled task                                  */
4              if Suc(τ) = ∅ ∨ SD_τ' = true ∀τ' ∈ Suc(τ) then
                   /* All successors have been scheduled or leaf
                      node.                                              */
5                  WaitingQueue ← WaitingQueue ∪ {τ};
6              end
7          end
8      end
9      WaitingQueue ← Sort_{P_τ}(WaitingQueue); /* Sort the waiting
           queue according to task's priority.                       */
10 return WaitingQueue;
```

$$f(\tau_i) = (N_{G_i} - 1) \times T(G_i) + CP(\tau_i) \tag{4.1}$$

If there are multiple tasks with the same priority, the task with the longest execution time will be selected. In the waiting queue, the priority value $P(\tau_i)$ of each task will be calculated according to Equation (4.2), where d_i represents the absolute deadline of the released task. Please note that τ_i here represents a task from DAG i (G_i). Within a single hyper period, a DAG could be released multiple times, and τ_i denotes a task from one of these releases. Additionally, N_{G_i} is computed for each release.

$$P(\tau_i) = d_i - f(\tau_i) \tag{4.2}$$

Then the waiting queue will be sorted in increasing order, and the task at the head of the queue will be selected and scheduled.

4.3.3 Mixed Schedule Calculation and Consistent Schedule Generation

In this subsection, the detailed procedure for mixed schedule calculation will be introduced, and the consistent schedule generated by "shrinking" the higher

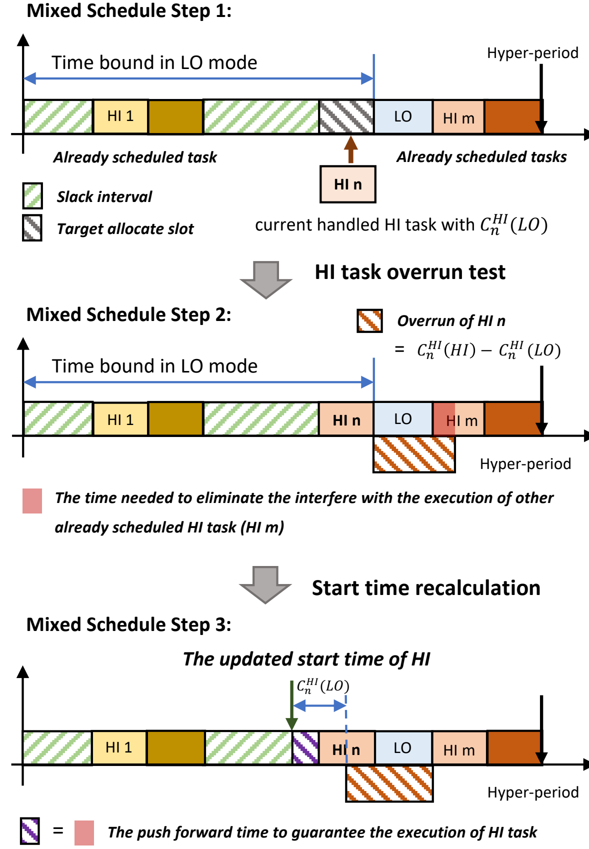

Figure 4.10: Task allocation on mixed schedule

criticality tasks will be proven safe. As Figure 4.10 illustrates, the mixed schedule calculation consists of three main steps.

Step 1. Task selection and slack interval analysis

The task with the highest priority in the waiting queue will be selected, for example, HI_n in the first graph of Figure 4.10 for a dual-criticality system. Based on the current status of the mixed schedule, as illustrated in step 1 of Figure 4.10, each available slack slot of the tested core (marked by green stripes) should be identified, and only slots larger than the minimum execution fragment length can be recognized as valid. Only if the sum of all valid slack slots is larger than the expected execution time can the latest slack interval be selected first (i.e., the time slot before LO in this example) to allocate the execution of the selected task. The end time of the selected time slot is also the expected end time of the task's execution.

The expected execution time of the selected task is determined by the criticality level of the task immediately following the latest allocated slack time slot. In this example, the task is LO marked by a blue box, working as a reference task, and the criticality of the selected task is HI, which is higher than LO. Therefore, the expected execution time is the one estimated in the system mode with criticality the same as the reference task, i.e., $C(LO)$. If the selected task has an equal or lower criticality level than the reference task, the expected execution time is the one estimated in the system criticality mode with the same value as the selected task itself. For example, if the criticality of the reference task is MI in a triple-criticality system, and the selected task is LO, then the expected execution time of the selected task is $C(LO)$.

Valid slack intervals of each core are determined based on the previously mentioned slack interval analysis. Assume τ_c is the reference task of the currently handled task τ_i on a specific core, based on the existing schedule, slack intervals can be analysed in the time duration $[0, \zeta_{\tau_c}]$, where ζ_{τ_c} represents the start time of τ_c. Suppose there are k valid slack intervals, and the sum of them I is larger than the expected execution time of τ_i according to Equation (4.3), we can start to calculate the start time ζ_{τ_i} of τ_i. C_{min} represents the minimum execution fragment time, and $L_{\mathcal{X}}$ should be selected according to Equation (4.4) to define the expected execution time $C^{L_i}_{\tau_i}(L_{\mathcal{X}})$ of task τ_i.

$$I = \sum_k I_k \geq C^{Li}_{\tau_i}(L_{\mathcal{X}}), \forall I_k \geq C_{min} \tag{4.3}$$

$$L_{\mathcal{X}} = \begin{cases} L_c & \text{if } L_i > L_c \\ L_i & \text{if } L_i \leq L_c \end{cases} \tag{4.4}$$

Step 2. Start time initialisation and overrun test

The start time calculation follows Equation (4.6), which is iteration-based because there could be time slots in the time interval $[\zeta_{\tau_i}, \zeta_{\tau_c})$ occupied by already scheduled tasks with criticality levels equal to or higher than the handled task, and their execution with $C(Li)$ should be guaranteed. η_{τ_k} represents the end time of task τ_k. Additionally, slack intervals I_f with a size smaller than C_{min} also force the start time to move earlier.

$$I_f = \sum_k I_k, \forall I_k < C_{min} \tag{4.5}$$

$$\zeta_{\tau_i} = \zeta_{\tau_c} - C_{\tau_i}^{L_i}(L_{\mathcal{X}}) - \sum_{\substack{\forall \tau_k \in (\zeta_{\tau_i} \le \zeta_{\tau_k} < \zeta_{\tau_c} \\ \bigvee \zeta_{\tau_k} \le \zeta_{\tau_i} < \eta_{\tau_k} \\ \bigwedge L_k \ge L_i)}} C_{\tau_k}^{L_k}(L_i) - I_f \tag{4.6}$$

In the multi-criticality systems example in Figure 4.10, the current handled task can be allocated to the target time slot with LO mode behaviour $C(LO)$. As the second graph of Figure 4.10 demonstrates, after finishing start time initialisation, the overrun test is vital to guarantee the schedule's safety. It can check whether, once an overrun happens, the already scheduled HI criticality task (*i.e.*, HI_m in this example) will be impacted. To guarantee the safety of the task schedule calculation, once the execution of HI task HI_n exceeds its LO execution time, we assume that it will reach its worst-case estimate $C(HI)$ and execution time difference $C(HI)-C(LO)$ is marked by red stripes. Then, the overlap between two HI criticality tasks can be calculated, which is marked by the red box. (Note that the impacted HI task could be the already scheduled HI task from the same core or the successors of the current handled task deployed on another core from the same DAG.) The impacted LO task can also be identified simultaneously.

For multi-criticality systems, overlap checking between tasks during the overrun in each system mode is required. For example, in a triple-criticality system ($S = HI, M, LO$), if the system is running with M criticality mode, all tasks with criticality levels equal to or larger than M should be guaranteed with their $C(M)$. Suppose the currently handled task is HI; we need to check whether, when it is executed with $C(M)$, there is any overlap between other already scheduled tasks also executed with their $C(M)$ in the same core or its successors with criticality level of M or HI. When the system moves to HI criticality mode, we must guarantee that there is no overlap between any two HI-criticality tasks. Generally, we need to avoid the execution of τ_i from affecting the already scheduled tasks with the same or higher criticality level (i.e., $L_h \ge L_i$), which are τ_i's successors or scheduled to be executed after τ_c on the same core and grouped into a task set Ψ_{hec}. If the start time ζ_{τ_h} of τ_h, where $\tau_h \in \Psi_{hec}$, is earlier than the end time η_{τ_i} of the current handled task τ_i iteratively calculated according to Equation (4.7), then the overall overlap detected for τ_i can be calculated following Equation (4.8), and the start time should be recalculated following the methods described in the next subsection. Please note that, in contrast to Equation (4.6), the current handled task τ_i is

executed with its WCET in the system mode, which has the same criticality level as the task itself $C_{\tau_i}^{L_i}(L_i)$.

$$\eta_{\tau_i} = \zeta_{\tau_i} + C_{\tau_i}^{L_i}(L_i) + \sum_{\substack{\forall \tau_k \in (\zeta_{\tau_i} \leq \zeta_{\tau_k} < \eta_{\tau_i} \\ \bigvee \zeta_{\tau_k} \leq \eta_{\tau_i} < \eta_{\tau_k} \\ \bigwedge L_k \geq L_i)}} C_{\tau_k}^{L_k}(L_i) + I_f \tag{4.7}$$

$$V_{\tau_i} = \sum_{\forall \tau_h \in \Psi_{hec}} \alpha \times (\eta_{\tau_i} - \zeta_{\tau_h}) \tag{4.8}$$

$$\alpha = \begin{cases} 1 & \text{if } \eta_{\tau_i} - \zeta_{\tau_h} > 0 \\ 0 & \text{if } \eta_{\tau_i} - \zeta_{\tau_h} \leq 0 \end{cases} \tag{4.9}$$

Step 3. Start time recalculation

Based on the time required to avoid conflict between two HI tasks (*i.e.*, the overlap time), the overall needed time can be obtained and recognised as "push forward" time. In the dual-criticality system's example illustrated in Figure 4.10, before allocating the LO task, if there exists sufficient overall valid slack interval greater than the sum of the LO execution time of the handled HI task and the "push forward" time, the final start time can be updated in the mixed schedule. If not, that implies the system is unschedulable.

When extending to a multi-criticality system, the overlap time can be analysed according to Equation (4.8), and the start time update should be discussed under two different conditions.

Condition 1: If $V_{\tau_i} \geq 0$, the start time of the selected task should be updated by Equation (4.10). Once the start time is updated, we need to analyse the time duration from the updated start time to the old start time $[\hat{\zeta}_{\tau_i}, \zeta_{\tau_i}]$ and identify if there are any scheduled tasks with the same or higher criticality with execution time $C_{\tau_k}^{L_k}(L_i)$ under the system mode with the same criticality level as the selected task τ_i. These tasks need to be involved in start time recalculation, as the third term of Equation (4.10) shows. The final start time of τ_i can be calculated and stay unchanged.

$$\hat{\zeta}_{\tau_i} = \zeta_{\tau_i} - V_{\tau_i} - \sum_{\substack{\forall k \in (\hat{\zeta}_{\tau_i} \leq \zeta_{\tau_k} < \zeta_{\tau_i} \\ \bigvee \zeta_{\tau_k} \leq \hat{\zeta}_{\tau_i} < \eta_{\tau_k} \\ \bigwedge L_k \geq L_i)}} C_{\tau_k}^{L_k}(L_i) + I_f \tag{4.10}$$

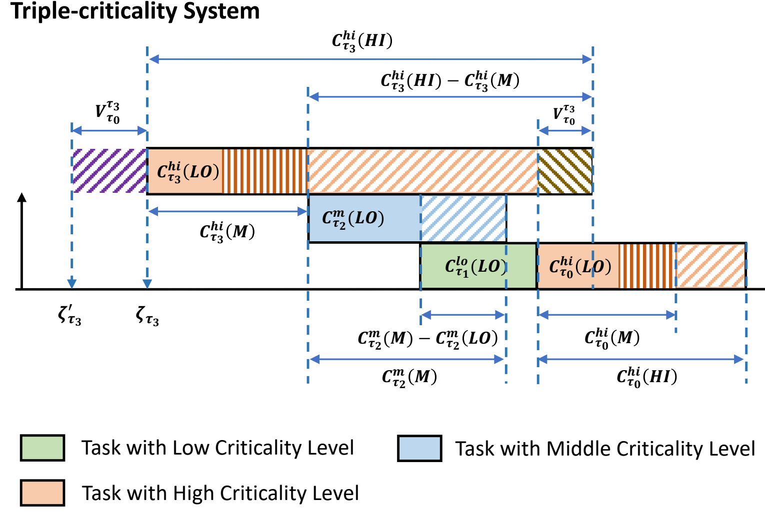

Figure 4.11: Example of task scheduling calculation in a triple-criticality system

After the start time is obtained, we can identify the tasks that may be impacted by τ_i in case of an overrun. The potentially impacted tasks are the ones that are scheduled in the time interval $[\hat{\zeta}_{\tau_i}, \eta_{\tau_i}]$, which are from the same core and with criticality values lower than τ_i. They are grouped into $\Gamma^{\tau_i}_{lc}$ and $\Gamma^{\tau_i}_{lc} = \{\tau_0, \ldots, \tau_{n-1}, \tau_n\}, L_\tau < L_i, \forall \tau \in \Gamma^{\tau_i}_{lc}$. Once the selected task overruns, the tasks with lower criticality need to be discarded but are expected to keep minimum execution time denoted by δ_j, where $j \in \Gamma_{lc}$.

Condition 2: If $V_{\tau_i} < 0$, the selected task τ_i will not jeopardize the execution of the already scheduled task with the same or higher criticality level, and all impacted tasks when facing τ_i overrun are with lower criticality levels. We don't need to update the start time and keep the results obtained according to Equation (4.6). Then, the effort should focus on analyzing the tasks scheduled in the time interval $[\zeta_{\tau_i}, \eta_{\tau_i}]$, where η_{τ_i} represents the end time of τ_i working with $C^{L_i}_{\tau_i}(L_i)$ and can be iteratively calculated according to Equation (4.7). Similar to condition 1, $\Gamma^{\tau_i}_{lc}$ represents a set of impacted tasks with lower criticality values than τ_i, which can be affected when τ_i overruns. The analysis time interval is updated by $[\zeta_{\tau_i}, \eta_{\tau_i}]$.

For both conditions 1 and 2, if task τ_i only overruns a small margin, the free time slot after the end time of τ_i and the start time of other tasks with equal or higher criticality values can be allocated to the task from task set $\Gamma^{\tau_i}_{lc}$. The task selection can start from the one with a higher criticality level.

Example: Figure 4.11 illustrates an example of task schedule calculation in a triple-criticality system (i.e., low mode (LO), middle mode (M), and high mode (HI)). Four tasks need to be scheduled on the same core: τ_0 and τ_3 are HI-criticality tasks with three estimated execution times under different system modes: $C^{hi}_{\tau_0}(LO)$ and $C^{hi}_{\tau_3}(LO)$ for the system with LO mode, $C^{hi}_{\tau_0}(M)$ and $C^{hi}_{\tau_3}(M)$ for the M system mode, and $C^{hi}_{\tau_0}(HI)$ and $C^{hi}_{\tau_3}(HI)$ for the system running in the HI criticality level. As the highest criticality tasks in the task set, their execution cannot be impacted by any other tasks. τ_2 is a task with M criticality with two execution time estimates: $C^{m}_{\tau_2}(LO)$ and $C^{m}_{\tau_2}(M)$. Its execution should be guaranteed in a system running with M mode. However, if the execution of any HI task will be affected by it, τ_2 must be discarded. If sufficient slack time exists, it is expected to execute δ_{τ_2} for data refreshing. τ_1 is a task with the lowest criticality level. Its execution only needs to be guaranteed under the LO system mode, and the estimated execution time is denoted by $C^{lo}_{\tau_1}(LO)$.

Assuming the scheduling order is $\tau_0, \tau_1, \tau_2, \tau_3$, which is determined based on the method introduced in Section 4.3.1 and Section 4.3.2, assuming τ_0 is the sink node from a DAG and there isn't any scheduled task in the tested core, then τ_0 can be directly allocated to the core as the first scheduled task. To guarantee its execution in HI system mode, the start time is calculated following $\zeta_{\tau_0} = T_{hyper} - C^{hi}_{\tau_0}(HI)$ without any further update, where $Thyper$ represents the hyper-period of all tasks.

After that, τ_0 works as the reference task for τ_1 and the available slack time interval for τ_1 is $[0, \zeta_{\tau_0}]$. The expected end time is the end point of the latest valid slack time interval. τ_0 has a higher criticality level than τ_1. Therefore, $C^{lo}_{\tau_1}(LO)$ is used to calculate the start time $\zeta_{\tau_1} = \zeta_{\tau_0} - C^{lo}_{\tau_1}(LO)$ following Equation (4.6). No scheduled task is located between ζ_{τ_1} and ζ_{τ_0} or ζ_{τ_1} is between the start and end time of any scheduled task so that the start time can be fixed.

Then, the slack time interval for τ_2 allocation is set with $[0, \zeta_{\tau_1}]$. τ_2 with m criticality, which is higher than τ_1 (i.e., $L_{\tau_1} = lo$). Therefore, $C^{m}_{\tau_2}(LO)$ is used for schedule calculation and analysing overlap between τ_2 and τ_0 when facing overrun is required because τ_0 is the task with a higher criticality level than τ_2 and scheduled after τ_1 from the same core. Without tasks with equals or higher criticality levels located between the start and end time of τ_2, the overall overlap for τ_2 can be obtained following Equation (4.8), i.e., $V_{\tau_2} =$

$\zeta_{\tau_2} + C^m_{\tau_2}(M) - \zeta_{\tau_0}$. As Figure 4.11 shows, there is no overlap and $V_{\tau_2} < 0$ so that the start time of τ_2 can be fixed and working as the reference task for τ_3.

Finally, according to Equation (4.8), we can find $V_{\tau_3} > 0$ and the method proposed for condition 1 should be followed. The affected task with the same or higher criticality is τ_0. Though τ_2, τ_1 are also impacted, with lower criticality levels, only the overlap time between τ_3 and τ_1 needs to be analysed for τ_3's start time updating and, as Figure 4.11 intuitively illustrates, the updated start time on the test core $\hat{\zeta}_{\tau_3}$ can be fixed.

Algorithm 7 illustrates the detailed pipeline to realise time stamp level schedule allocation. To satisfy criticality-dependent timing requirements, we use the schedule in the corresponding criticality system mode to identify the free time stamp and allocate it to the task working with its worst-case execution time in its highest criticality level. In the system mode with a criticality value equal to the highest criticality level of the handled task, all tasks are working with their worst-case execution time (WCET) in the same mode. Therefore, once the conflicts between tasks have been eliminated, the safe execution of the handled task can be guaranteed.

Algorithm 7: Schedule On Core Calculation

```
1  Function ScheduleOnCore(τ, m, D):
2      if D = true then
3          Dry run: Only calculate and return result, schedule E and
             E(S) remain unchanged;
4          E_backup ← E;
5          E(S)_backup ← E(S);
6          ζ_backup ← ζ;
7      end
8      ζ_τ^m ←ScheduleOnCoreInModeS(τ, m, L_τ);
9      if D = true then
10         Dry run: revert changes;
11         E ← E_backup;
12         E(S) ← E(S)_backup;
13         ζ ← ζ_backup;
14     end
15     SD_t ← true;
16     return ζ_τ^m;
```

Algorithm 8: Schedule On Core In S criticality system mode

```
1  Function ScheduleOnCoreInModeS(τ, m, L_τ):
2      S = L_τ; /* Use the schedule in accordance with the
           criticality system mode that aligns with the task
           that needs to be scheduled.                              */
3      η^s_ed ← P^τ_ed; /* The latest end time of task τ, determined
           by task dependency and is initialized with τ's
           period end time.                                         */
4      ζ^s_st ← T^τ_st; /* The earliest start time of task τ,
           determined by task dependency and is initialized
           with τ's Period start time.                              */
5      if Suc(τ) ≠ ∅ then
6          η^s_ed ← min ζ_τ', {∀τ' ∈ Suc(τ)} ∧ {L_τ' ≥ L_τ};
7      end
8      I = InValidIntervalIdentification(ℰ(S));
       /* Mark time stamps from invalid free time slots as
           occupied.                                                */
9      if I ≥ C^{L_τ}_τ(L_τ) then
10         exe ← C^{L_τ}_τ(L_τ); /* execution time of task in the
               system mode with the same criticality level         */
11         ζ^m_τ ← TimeStampAllocation(τ, m, L_τ, η^s_ed, exe, η^s_ed, ζ^s_st);
12         return ζ^m_τ;
13     end
14     else
15         return Unschedulable;
16     end
```

Step 4: Core selection and schedule update

For a multi-core platform, the core selection process is based on testing. Following the same procedure from the start time calculation steps (step 1 to step 3), the start time of the handled task τ_i on different cores can be calculated as $\zeta^{\iota}_{\tau_i}, \forall \iota \in N$, where N represents the number of cores. The core providing the latest start time will be selected according to Equation (4.11) to offer more flexibility to the unscheduled task. On an N-core platform, the core with the lowest workload (i.e., the core with the least amount of scheduled

Algorithm 9: InValid Interval Identification

```
1  Function InValidIntervalIdentification(ℰ(S)):
2      for ts = hyperperiod : 1 do
           /* Start from the last time point of hyperperiod   */
3          if ℰ^m_ts ≠ ∅ ∧ ℰ^m_{ts−1} = ∅ then
4              ζ_temp = ts - 1;
5          end
6          if ℰ^m_ts = ∅ ∧ ℰ^m_{ts−1}¬∅ then
7              η_temp = ts - 1;
8          end
9          if ζ_temp − η_temp ≤ C_min then
               /* If the interval is smaller than the minimum
                  execution fragment, all timestamps within that
                  interval will be marked as occupied.        */
10             for ts = ζ_temp : η_temp do
11                 ℰ_t s(S) ← Occupied;
12             end
13         end
14         else
15             I+ = ζ_temp − η_temp;
16         end
17     end
18     return I;
```

Algorithm 10: Time Stamp Allocation

```
1  Function TimeStampAllocation(τ, m, L_τ, exe, η^s_ed, ζ^s_st):
2      for ts = η^s_ed : ζ^s_st do
           /* Allocate time stamp to the selected task τ in a
              backwards direction.                                  */
3          if E^m_{ts-1} ≠ ∅ ∧ E^m_{ts} = ∅ then
4              τ_c ← E^m_{ts-1};
5          end
6          if exe > C^{L_τ}_τ(L_{τ_c}) then
7              Scheduling overrun execution timestamp;
8              if E^m_{ts} = ∅ then
9                  exe ← exe − 1;
10                 E^m_{ts}(L_τ) = τ;
11             end
12         end
13         else
14             Scheduling C^{L_τ}_τ(L_{τ_c}) execution timestamp;
15             if E^m_{ts}(L_τ) = ∅ ∧ E^m_{ts} = ∅ then
16                 exe ← exe − 1;
17                 E^m_{ts}(L_τ) = τ;
18             end
19         end
20         ts_last ← ts; /* The last scheduled time stamp is the
              start time of the currently handled task τ.          */
21         if exe = 0 then
22             break;
23         end
24     end
25     if exe > 0 then
26         return Error; /* Unschedulable on this core              */
27     end
28     else
29         ζ^m_τ ← ts_last;
30         return ζ^m_τ;
31     end
```

tasks) will be selected when faced with the same start time, as indicated by Equation (4.12). Finally, the selected core number ι and the start time of τ_i on it can be determined. Simultaneously, the schedule is updated, and task τ_i can be removed from the waiting queue (i.e., the queue of unscheduled tasks waiting for execution), which is then updated by adding the successors of τ_i.

$$\zeta^{\iota}_{\tau_i} = max(\zeta^{m}_{\tau_i}), \forall m \in N \tag{4.11}$$

$$\iota = ind(min(U_m)), \forall m \in N \tag{4.12}$$

The utilisation value of the selected core should be updated according to $U_\iota \leftarrow U_\iota - \frac{C_{\tau_i}}{T_{hyper}}$. Here, C_{τ_i} represents the execution time of the allocated task, and T_{hyper} is the hyper-period of the system. For both HI and LO tasks, $C_{\tau_i} = C_{\tau_i}(LO)$ because the final consistent schedule used for system maintenance in normal mode is $C(LO)$-based. When extending a multi-criticality system, the utilisation value updating of the selected core is based on its execution time under the lowest criticality system mode $C^{L_i}\tau_i(S_0)$ following Equation (4.13), where S_0 represents the lowest system mode. $U\iota$ represents the remaining utilisation of the selected core ι.

$$U_\iota \leftarrow U_\iota - \frac{C^{L_i}_{\tau_i}(S_0)}{T_{hyper}} \tag{4.13}$$

After that, the main pipeline is repeated to select the next unscheduled task with the highest priority from the updated waiting queue, and the process is continued until either the waiting queue is empty or an unschedulable error occurs.

4.3.4 Consistent Schedule based Task Management At Run Time

Based on the proposed method in the previous section, the start times of tasks with different criticality levels can be fixed in a consistent schedule. In this section, we use a consistent schedule example to explain how to maintain the execution of tasks at runtime and achieve task-level mode changes.

Assume that, according to our proposed method, a consistent schedule for a dual-criticality system can be obtained, and Figure 4.12 illustrates one hyperperiod schedule. All tasks are working ideally with their $C(LO)$. Figure 4.13 illustrates the worst-case scenario, in which all HI criticality tasks overrun. Based on the consistent schedule, it is not difficult to find that the

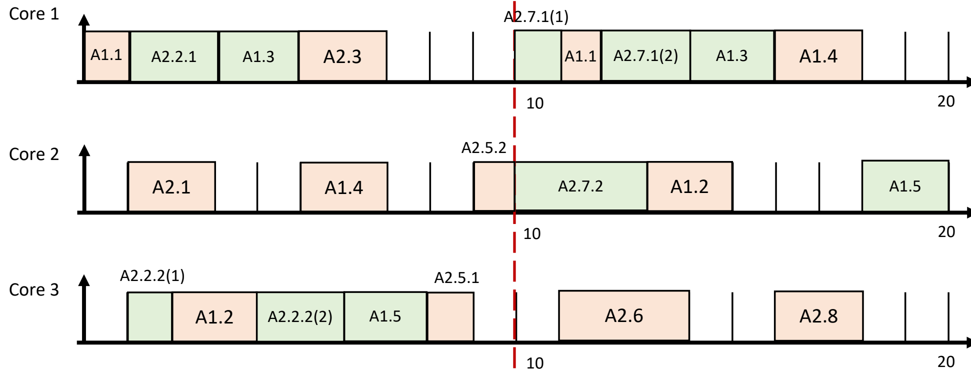

Figure 4.12: The consistent schedule example for a dual-criticality system

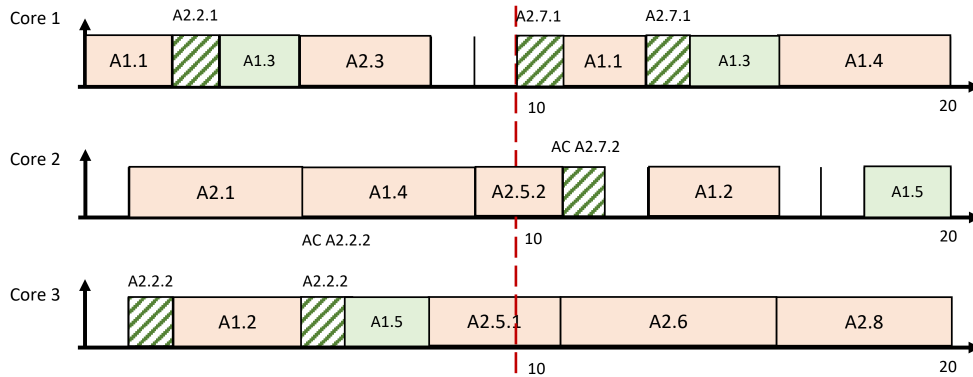

Figure 4.13: The consistent schedule example when all HI tasks overrun

overrun of each HI task can only interfere with the execution of specific LO tasks, which have been clustered into LO task groups corresponding to the specific HI tasks during schedule calculation. Furthermore, there exist LO tasks that will never be discarded, such as $A1.5$ and $A1.3$.

If only $A1.1$ overruns, it leads to the performance degradation of $A2.2$, which consists of two sub-functions, $A2.2.1$ and $A2.2.2$. Only $A2.2.1$ is impacted, while $A2.2.2$, deployed on core three, can still function normally. Suppose that before the start of $A1.3$, there is sufficient time to support minimum operation. In this case, $A2.2.1$ will not be fully discarded and can continue working in a degraded mode. The system can then transition to the HI mode. However, this only means that all tasks with the same or higher criticality level as the system mode (i.e., HI) are allowed to run at their maximum estimated WCET ($C(HI)$), while tasks with lower criticality values can be discarded or degraded to free up resources and guarantee the execution of tasks with

higher criticality when necessary. Unaffected tasks can continue their normal execution. In this case, we only need to monitor the actual execution time of each HI task and discard or degrade the affected LO tasks as needed.

It is worth noting the execution of $A2.2.2$. Its start time is earlier than that of the HI task $A1.2$. According to the static schedule, its execution would be preempted by $A1.2$. Let us assume $A1.2$ overruns (i.e., it cannot finish its execution within the estimated execution time), $C(LO)$), which implies that $A2.2.2$ cannot resume in time. For instance, in a ROS2 node, $A2.2.2$ might not provide valid results before the start of $A1.5$. However, $A2.2.2$ has been partially executed, and the minimum operation (such as data refreshing) can be considered finished. As a result, it can be classified as a degraded node.

4.4 Evaluation

The evaluation of this work consists of two parts. First, we use a real-world application deployed on a robot to prove the necessity of the minimum execution concept involved in our proposed scheduling method. We use ROS 2, a robot operating system, and Python to implement all related functions. Although the code in our work is not optimized, the observed performance can prove the necessity of the minimum execution concept involved in our proposed scheduling method (i.e., keeping LO criticality tasks in degraded mode). The second part of the evaluation uses synthetic task sets to assess the feasibility and efficiency of our proposed static scheduling method. The advantage of using synthetic tasks, compared to using a single example, is that we can explore system behaviour under various different conditions.

4.4.1 Experiment on a Real-world Application

A Camera-LiDAR fusion-based object detection algorithm was deployed on a mobile delivery robot to verify the necessity of keeping minimum execution for data refreshing. We used Robot Operating System (ROS2) and Python to integrate the algorithms because the start and stop time of each node can be controlled individually, which can be considered as restart and discard in this work. Based on ROS2, if a task node is discarded, it is essentially terminated and requires restarting. If a task node is degraded, it implies that the node only maintains data refreshing and bypasses the processing operations. Additionally, the executor tool was used to assist us in determining

the predecessor constraints between different nodes. The Nvidia Jetson AGX Xavier serves as the computational platform for our experiments. To assist in realising object detection, we use one depth camera (Intel RealSense Depth Camera D435i) and one single-line scanning laser rangefinder (Hokuyo UST-10LX) to provide sensor data.

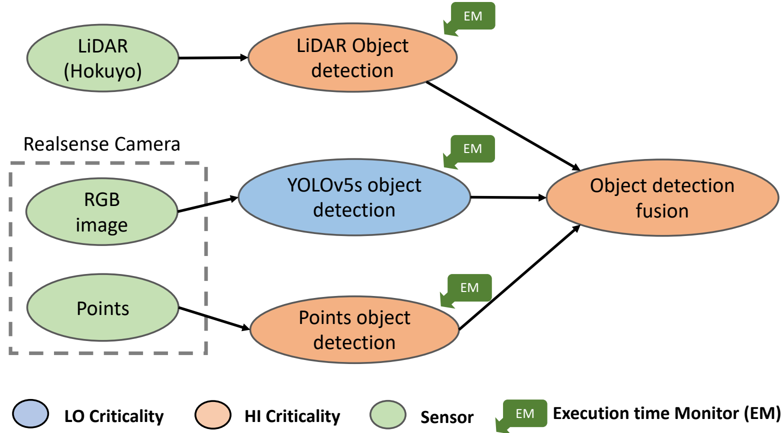

Figure 4.14: The structure of object detection function

Table 4.1: Time consumed by YOLOv5s and points nodes in different scenarios.

Node	Scenario	Max(s)	Min(s)	Mean(s)	Median(s)	Standard Deviation
YOLOv5s	Lab	0.057	0.032	0.046	0.045	0.004
	Living room	0.064	0.037	0.045	0.045	0.004
Points	Lab	0.468	0.147	0.239	0.235	0.036
	Living room	1.367	0.279	0.412	0.447	0.094

Figure 4.14 illustrates the object detection deployed on a mobile robot. Compared with cameras and infrared sensors, single-line LiDAR can provide stable data at longer distances and a wider field of view. It is used to ensure the safety of the robot, and the object detection function based on it is recognized as a HI criticality task. YOLOv5s [75] is utilised to realise RGB-image-based object detection. It is easily impacted by illumination conditions, and the accuracy highly depends on the selected dataset for model training. Because of its relatively lower reliability level, the YOLOv5s node is regarded as an LO criticality task. The points data provided by the camera is based on the

fusion of the camera and infrared sensors. We assume it can provide relatively stable sensor data and use a clustering-based method (i.e., DBSCAN [116]) for short-range object detection to reduce the risk caused by objects that cannot be detected by the single-line scanner. Therefore, the points detection node is also determined as a HI criticality task.

In any driving scenario, the execution of HI criticality tasks should be guaranteed. Table 4.1 summarizes the recorded execution times of the YOLOv5s and Points object detection nodes in 1000 frames in different scenarios with diverse complexity levels. In the lab environment, the obstacles are predefined with a simple and clean background. On the contrary, the living room environment is much more complex, with objects distributed in different locations. Furthermore, the number, shape, and size of objects are not fully certain. As a state-of-the-art algorithm designed for real-time detection, the execution time of the YOLOv5s node does not face significant execution time variation. However, the execution time of the clustering algorithm varies significantly because the density of raw data released by the RealSense camera in different scenarios changes considerably. Some parameters need to be adapted to balance the accuracy and execution time (e.g., down-sampling rate, $min_samples$, and eps for DBSCAN).

To guarantee the execution of Points and LiDAR detection, which are determined as HI criticality tasks in our solution, the YOLOv5s node can be discarded. Runtime monitors can monitor the execution time of each node. If the execution time of the Points or LiDAR detection node exceeds the predefined threshold, this is regarded as an overrun, and the YOLOv5s node needs to be discarded. However, in our work, minimum execution for data refreshing is expected. The cost of restoring the YOLOv5s node from a discarded state and a degraded state is recorded in Table 4.2, which is created based on 100 trials. We can observe that the restoration time from the discarded state is around 40 times slower than the time consumed from the degraded state. This is caused by the overhead of PyTorch module reloading after re-initialization. In practice, the delay of restarting a task would delay the whole reaction time of the system from recovering from the HI-mode back to the LO-mode. Although for other applications, the restoration time would be much shorter, we note that this can still be 2-10 times higher compared to resuming from a degraded state due to, for example, cache.

Table 4.2: Time consumed by YOLOv5s node restoration from different initial states.

Initial State	Max(s)	Min(s)	Mean(s)	Median(s)	Standard Deviation
Discarded	32.417	30.901	31.4741	31.4745	0.405
Degraded	0.930	0.655	0.740	0.742	0.047

4.4.2 Simulation-based Experiment

We present results produced using a simulation implemented in Python. To limit complexity, we use a dual-criticality system to evaluate our proposed method, as it is sufficient to evaluate the efficiency and feasibility of our method. The *experiment setup* has the following phases:

DAG generation: We use an existing DAG generator [43] to simulate DAGs, where the maximum depth (the number of layers) is randomly selected from 4 to 6, and the number of nodes in each layer is uniformly distributed from 2 to 8. The scheduling calculation duration is set to one hyper-period and repeated at runtime. Each DAG may have different periods, and during scheduling calculation, DAGs may be released multiple times. To limit the complexity, we assume that the number of DAGs in each system is no more than 4.

Execution time generation: After fixing the DAG structure, we use *UUnifast* [53] to synthesize the execution time and period of each task under different system utilisations. To limit complexity, we assume that there are only three cores in the system, and thus the sum of the utilisation of each task $U_{system} = \sum C_i(LO)/T_i$ cannot be greater than 3. The normalized utilisation (i.e., U_{system}/m, where m is the number of cores) variation of large-scale generated systems is from 0.3 to 1.0. For period selection, tasks from the identical group (DAG) inherit the DAG's period, which is randomly selected from a predefined set [100, 200, 500, 1000, 2000, 5000, 10000, 20000, 50000], reflecting the empirical values widely used in the real-time system community for complex (e.g., autonomous) systems. For the execution time generation, we assume that LO tasks have a single worst-case execution time estimate $C_l(LO)$, but with a one-time unit minimum operation. $C_l(HI) = 1$. HI-criticality tasks have dual execution time assignments, and $C_h(HI) = CF \times C_h(LO)$, where CF (Criticality Factor) is uniformly randomly selected between 1.5 and 2. It

is worth noting that the system utilisation after a mode change cannot exceed 3 (i.e., $\sum C_i(HI)/T_i \leq m$).

HI criticality task selection: In the generated graph, the sink node of each application is regarded as a fusion node, which should be set as HI criticality. At least one node from the parent set of any HI criticality task should also be set as HI and randomly selected. Following this rule, we can find at least one end-to-end HI path to guarantee basic safe operation. In our work, the criticality proportion C_p is set to the default value of 0.5. If the number of tasks in the HI path is no larger than $C_p \times N$, where N represents the total number of tasks in the system, we will start from the source node and randomly select one of its *LO*-criticality successors to convert it to a HI task. The procedure will stop when $N_{hi}/N \geq 0.5$, where N_{hi} denotes the number of HI tasks in the system, and the remaining tasks are classified as *LO* criticality.

Large-scale system generation: after generation rules are decided, three system types consisting of 2, 3 and 4 DAGs were generated at scale. For each category, 5,000 systems were generated under each normalised utilisation. We have eight different workloads, and for each workload, 5,000 systems were generated, and each system comprises two different DAGs. Thus, 40,000 systems with 80,000 DAGs were generated for the 2-DAG system type. Overall, 120,000 systems with 360,000 DAGs were generated for evaluation.

4.4.3 Evaluation of Schedulability

We evaluate our method against two state-of-the-art methods referred to as *fed* [17] and *lsai* [92]. With *fed*, in high mode, tasks are scheduled with a non-preemptive strategy and the *LO* schedule is calculated based on fixed priority preemption. Because HI tasks always have higher priority than any *LO* tasks, their execution in *LO* mode can be guaranteed by preempting *LO* tasks. However, according to their strategy, more than one core would be occupied by a heavy load DAG and cannot be used by any other tasks from any other DAGs, which may lead to severe resource waste and reduce the schedulability of the systems. For example, if the utilisation of one heavy DAG equals 1.01, two cores will be occupied (as $ceil(1.01) = 2$), making it more likely that the system is unschedulable.

Another issue raised by the *fed* method is that HI tasks are not allowed to be dependent on *LO* tasks. However, a more practical system model is that

a HI task can depend on a LO task if there is at least one HI task in its predecessor set. Unfortunately, the *fed* method does not support this. In low mode, a HI task cannot be executed before the LO predecessor task finishes its execution, and that may lead to an unsafe mode change because the start time of the HI task in the HI schedule would be earlier than its start time in the LO schedule. Therefore, once an overrun happens, the remaining budget in the HI schedule cannot guarantee the execution of the HI task. In our approach, a system with an unsafe schedule will be regarded as unschedulable. As Figure 4.15 shows, under our system model assumption and safe operation check, even for a lightly loaded system consisting of two DAGs, the schedulability of *fed* is only 20%.

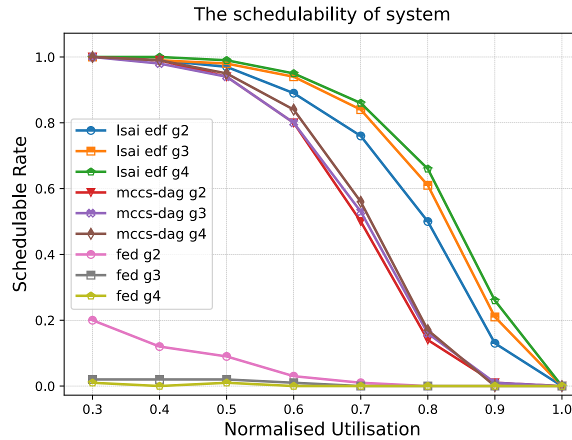

Figure 4.15: The schedulability of systems against normalised utilisation (g2, g3, and g4 represent systems comprised of 2, 3, and 4 DAG applications, respectively.)

To reduce the unschedulable rate, *lsai-edf* proposes a fully preemptive scheduling method and schedules HI tasks as late as possible (i.e., starting from the sink node of each DAG and allocating the execution time from the last time point of the hyper-period). In LO mode, all tasks are scheduled as early as possible from the source node and zero time instant of each hyper-period. This method can provide a much higher schedulable rate. However, time unit level calculation (i.e., after allocating one time unit, the priority of each ready task should be recalculated) may lead to a severe preemption rate. Moreover, tasks can be migrated to any core to get higher resource utilisation. In the worst case, two tasks can be preempted by each other every time unit, triggering frequent migration. During their schedule calculation,

the *Safe Trans. Prop.* strategy can guarantee the safety of the schedules for HI, LO mode, and during a mode change. Therefore, their method is compatible with our system model. As expected, a fully preemptive method can get better schedulability, especially under high loads.

In our work, we adopt the idea of a limited preemption strategy to get better schedulability, and limiting minimum execution fragments can significantly decrease the preemption overhead. Besides, we do not allow task migration at run-time. Inevitably, the schedulable rate is smaller than *lsai-edf*, as Figure 4.15 demonstrates.

4.4.4 Evaluation of Preemption and Migration Overhead

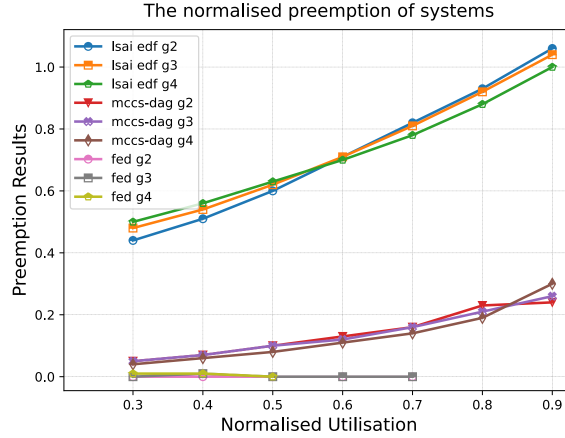

Figure 4.16: The normalised preemption rate of systems against normalised utilisation

Preemption and migration analysis use normalized preemption P_n and normalized migration M_n as metrics, calculated using $P_n = \phi_p/||S||$ and $M_n = \phi_m/||S||$. Here, ϕ_p and ϕ_m represent the number of preemptions and migrations within one hyper-period based on LO schedules , and $||S||$ denotes the number of tasks in the system. ϕ_p and ϕ_m can be obtained during the static schedule calculation.

Although the schedulability of *fed* is low and direct comparison is unfair, the preemption and migration analysis based on successfully scheduled systems shows their advantage from a preemption and migration perspective. They have different schedules for HI and LO modes; in HI mode, tasks are scheduled non-preemptively. The source of their preemption and migration

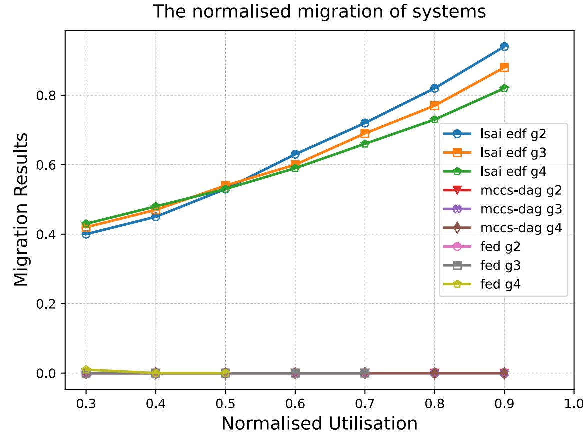

Figure 4.17: The normalised migration rate of systems against normalized utilisation

costs is the LO schedule. As Figure 4.16 shows, the normalized preemption value is close to zero in the scheduled system. In a lightly loaded system, all tasks can be scheduled in any system mode even with no preemption.

As noted in Sec.4.4.3, *lsai-edf* inevitably gives a high preemption and migration rate. The limited preemption and prohibition on migration in our work help to get normalized preemption values which are about 50% smaller than *lsai-edf* in systems with low utilisation (see Figure4.16). In a highly loaded system, although the schedulable rate of our proposed method becomes relatively lower, the normalized preemption value can be 60% smaller on average. For migration evaluation, as illustrated in Figure 4.17, the normalized migration value of *lsai-edf* increases linearly with the normalized utilisation.

4.4.5 Evaluation of Survivability

One of our key contributions is the ability to maintain a consistent schedule that ensures the safe execution of tasks in any system mode. In addition, we consider the survivability of LO criticality tasks during the schedule calculation process. This approach can save more LO tasks or at least degrade them, thus improving the system's resilience and QoS, even under the worst-case operation scenario (i.e., when all HI tasks are overrunning with their $C_i(HI)$). Task-level mode change eliminates the cost caused by the system mode change and provides more opportunities to preserve LO tasks.

To evaluate survivability, we use the average percentage of rescued and

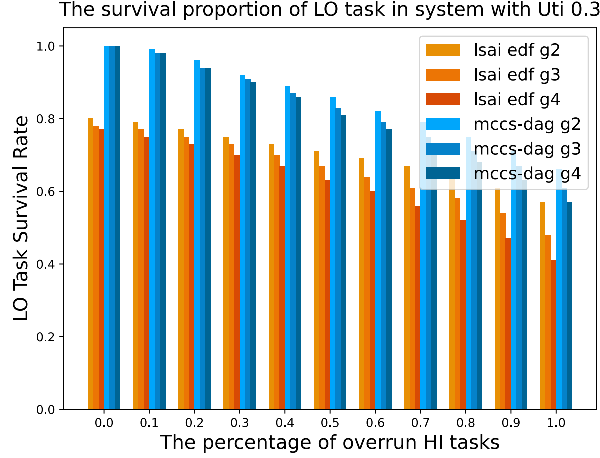

Figure 4.18: The survival proportion of LO tasks in systems with Uti=0.3

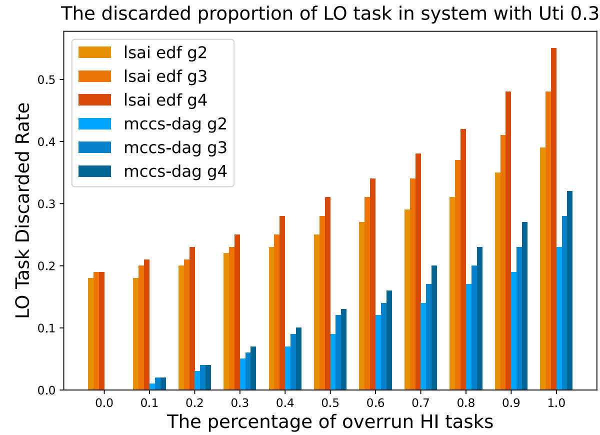

Figure 4.19: The discarded proportion of LO tasks in systems with Uti=0.3

discarded LO tasks of different system types under different system loads with the increasing number of overrun HI tasks. For each round, we randomly selected a specific percentage of the HI tasks to overrun (e.g., on the x-axis of Figure 4.18, 0.1 means that we randomly selected 10% of the HI tasks to overrun). We only use the HI schedule of *lsai-edf* for the comparison because the *fed* based method can not provide sufficient valid schedules to support a fair comparison. Based on the HI schedule, we attempt to insert LO tasks into the free time slots based on the actual execution time of each HI task. According to the system mode change rule, even though the execution time of a single HI task exceeds 0.1% of its $C_i(LO)$, the schedule should be changed

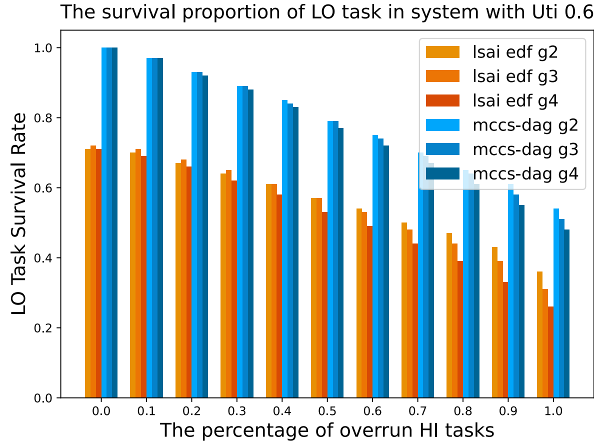

Figure 4.20: The survival proportion of LO tasks in systems with Uti=0.6

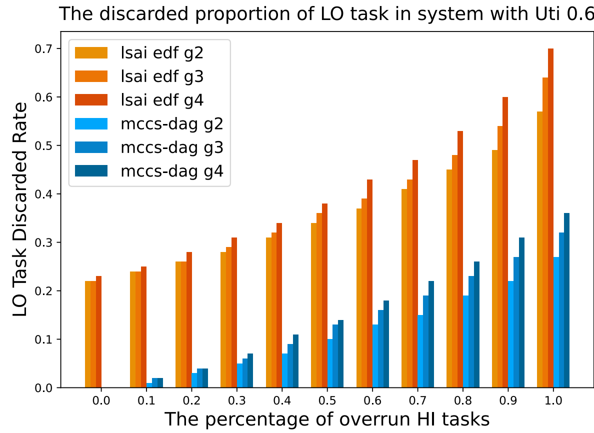

Figure 4.21: The discarded proportion of LO tasks in systems with Uti=0.6

to HI mode. Therefore, it is possible to find some time slots to insert LO tasks and satisfy the precedence requirement. For LO task insertion into the HI schedule of *lsai-edf*, we keep their fully preemptive strategy inserting LO task wherever possible. If LO tasks can not be fully inserted into the HI schedule, we also seek to find one time unit to provide minimum operation before the execution of its successors.

The evaluation results of systems with normalised utilisation of 0.3, 0.6, and 0.9 are selected to intuitively demonstrate the proposed method's performance in systems with low, medium, and high workloads. As shown in Figure 4.18, 4.20, 4.22, 4.19, 4.21 and 4.23, in all cases, our proposed method

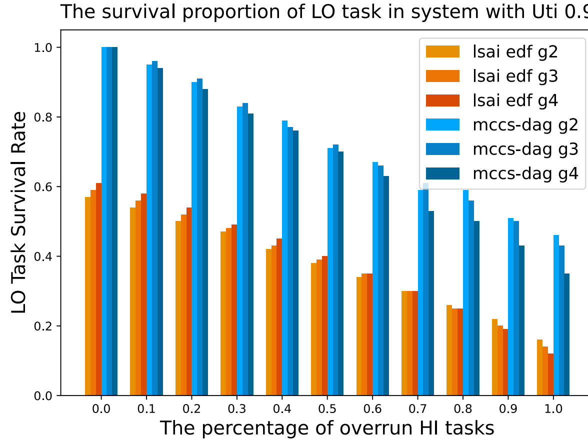

Figure 4.22: The survival proportion of LO tasks in systems with Uti=0.9

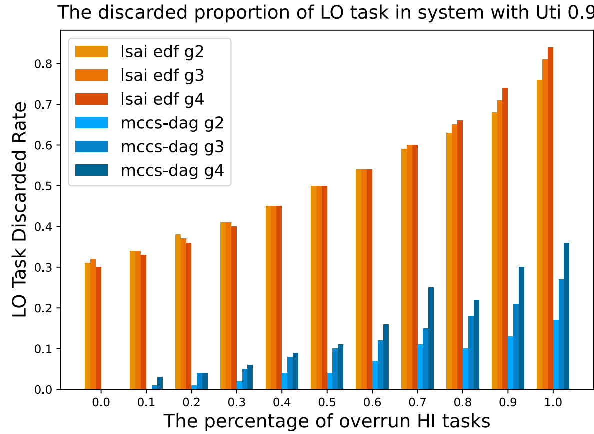

Figure 4.23: The discarded proportion of LO tasks in systems with Uti=0.9

always has a higher survival rate and lower discarded percentage than *lsai-edf*. In the best case, if only one HI task exceeds one time unit of its $C(LO)$, no mode change will happen based on our proposed consistent schedule, and no LO task will be discarded. However, almost 20% of LO tasks in the *lsai-edf* schedule-based system will be discarded because of the schedule update, even in systems with low workload (i.e., uti = 0.3). The percentage of survived tasks decreases with an increase in the number of overrunning HI tasks. In the worst case (i.e., the most complex system with four DAGs and a high workload with Uti = 0.9), we can still save 34% of LO tasks. It is worth noting that, based on the minimum operation strategy, the number of degraded

Table 4.3: The performance of proposed mccs-dag scheduling strategy compared with the lsai-edf method.

utilisation		Survival Rate Difference (%)			Discarded Rate Difference (%)			Degraded Rate Difference (%)		
	Overrun Tasks (%)	2-Graphs	3-Graphs	4-Graphs	2-Graphs	3-Graphs	4-Graphs	2-Graphs	3-Graphs	4-Graphs
0.3	10	19.65	20.72	22.08	-17.62	-18.17	-18.93	26.51	17.85	15.32
	50	14.5	15.97	17.96	-15.7	-16.43	-17.8	22.01	16.38	13.81
	90	10.12	13.05	15.59	-15.13	-17.34	-20.54	21.35	17.48	17.43
0.4	10	21.7	22.33	24.02	-18.91	-19.19	-20.41	29.82	17.99	18.54
	50	16.4	17.95	19.61	-17.81	-18.37	-19.46	24.51	18.0	14.88
	90	11.67	14.74	17.87	-17.97	-19.98	-23.57	24.13	19.6	18.81
0.5	10	23.83	24.62	25.78	-20.04	-20.91	-21.5	30.58	23.3	16.39
	50	17.71	19.77	21.56	-19.58	-20.35	-21.42	27.57	19.67	16.48
	90	13.69	17.73	19.72	-20.7	-23.66	-26.17	25.27	21.03	19.89
0.6	10	27.52	26.73	27.72	-22.69	-22.24	-22.92	37.27	25.64	20.04
	50	22.12	21.85	23.59	-24.36	-22.86	-23.92	32.08	23.12	19.5
	90	17.99	19.62	21.86	-27.5	-27.21	-29.7	30.7	23.81	22.2
0.7	10	31.19	30.34	30.15	-24.89	-24.86	-24.81	41.0	30.67	27.98
	50	24.85	25.25	25.68	-28.63	-27.44	-27.01	37.5	29.02	23.97
	90	21.75	23.09	23.96	-33.36	-33.24	-33.72	34.29	28.59	25.35
0.8	10	35.47	33.57	33.21	-27.76	-27.3	-27.39	44.54	41.24	34.34
	50	28.78	26.47	27.89	-34.87	-31.68	-31.59	45.71	36.77	32.11
	90	25.44	25.18	26.24	-41.31	-39.96	-39.51	41.12	36.11	31.12
0.9	10	41.8	40.28	35.81	-34.07	-32.55	-29.28	63.96	43.39	23.62
	50	34.55	33.7	29.58	-45.78	-40.68	-38.73	63.11	47.13	46.92
	90	30.64	29.65	23.34	-55.08	-49.47	-43.82	58.36	45.85	39.2

tasks also increases with the number of overrun tasks. Even in the worst case, the strategy can help us keep the discarded percentage relatively low, at 37%. Furthermore, once the system is back to normal mode, only 37% of LO tasks need to be reloaded, compared to 84% with *lsai-edf*. That means the system would take much less time to recover to normal mode.

In order to provide a more comprehensive evaluation result, Table 4.3 summarizes the overall performance of systems with different levels of structural complexity and workload. As emphasised before, except for the survival task rate, the degradation in this work means minimum execution for data refresh and less restoration than restoring a discarded task. The survival and discarded rate difference values are obtained as follows: $D_{sur} = R_{sur}^{mccs} - R_{sur}^{lsai}$ and $D_{dis} = R_{dis}^{mccs} - R_{dis}^{lsai}$, where D_{sur} and D_{dis} represent survival and discarded rate difference, respectively. R_{sur}^{mccs} and R_{dis}^{mccs} denote the survival rate and discarded rate of our proposed $mccs - dag$ method. R_{sur}^{lsai} and R_{dis}^{lsai} represent the performance of $lsai$. The standardized degraded rate difference

D_{deg} is calculated using Equation (4.14), which represents the difference of degraded rates among non-survived rates of each method. Please note that survival refers to a task being executed normally without any impact in any system operational mode. If a task is degraded, it is not considered as having survived but rather is running in a degraded mode to minimize data-refresh-related operations. Consequently, we utilize the proportion of degraded tasks among the non-survived tasks to assess the effectiveness of system recovery.

$$D_{deg} = (R_{deg}^{mccs}/(1 - R_{sur}^{mccs})) - (R_{deg}^{lsai}/(1 - R_{sur}^{lsai})) \tag{4.14}$$

We can observe that with the increase in system utilisation, the survival rate difference also grew, which means that our proposed method has better resilience than *lsai*. Besides, in any condition, discarded rate difference values are always smaller than zero, meaning fewer tasks need to be restored from the discarded state, and the system recovery time can be reduced. Furthermore, the higher standardised degraded rate difference means that our proposed method can provide much higher system recovery efficiency. More tasks restored from a degraded state suggest shorter time shall be consumed for system recovery.

In summary, our proposed method slightly sacrifices schedulability to reduce the preemption and migration cost. However, the schedulable rates are still acceptable under different workloads. Furthermore, a consistent schedule can ease the maintenance of mixed-criticality systems by eliminating system mode change. With the consideration of LO tasks' survivability during schedule calculation, a consistent schedule can enable task-level mode change and keep the execution of LO tasks longer. The minimum operation works as the degraded operation and assists in speeding up system recovery.

4.5 Summary

In this work, we proposed a novel mixed-criticality multi-core DAG scheduling strategy. Existing static scheduling methods generate two different schedules for different system modes and discard all LO criticality tasks in HI mode. In comparison, our strategy generates one consistent schedule considering the survivability of tasks with lower criticality levels to realise task-level mode change and significantly improve system resilience. Different schedules require more analysis effort to guarantee safety during mode change, and system re-

covery is non-trivial, potentially delaying the restoration of dropped tasks unnecessarily. Additionally, lower critical task allocation in the higher-level mode schedule becomes more complex due to precedence constraints between tasks with different criticality levels. As demonstrated in the evaluation, our method reduces complexity. Task-level mode change can accelerate the recovery of specific impacted tasks, and the existence of lower critical tasks also improves the efficiency of computational resource utilisation. The proposed methods can be extended to support event-triggered tasks, which can be executed using server-based methods to contain its timing behaviour. In this context, a server is recognized as a periodically released task and can be statically scheduled, with a reserved time slot for the execution of event-triggered tasks.

Chapter 5

Resilient and Efficient Time-Sensitive Network

This chapter extends the design of scheduling strategies from the task-level to the network-level. Time-Sensitive Networking (TSN) is being widely investigated to provide Ethernet capabilities for in-vehicle backbone communication. However, the gate control list (GCL), which is a simple mechanism for achieving timing determinism for safety-critical traffic (ST) frames with hard deadlines, is too rigid to handle the intrinsic timing uncertainty of automated driving systems (ADS). Due to the complexity and unpredictable operating environment, delayed ST frames can disrupt the solidly fixed timing behaviour. This chapter presents a novel approach to effectively use bandwidth resources to deal with delayed ST frames that a traditional fixed GCL cannot handle. Discarding such frames should be reduced to improve the system's integrity. An acceptance test is implemented to report to the application layer when an ST frame will miss its deadline and hence be rejected, i.e. prevented from entering the switch. To further improve the efficiency of bandwidth usage, we investigate how to improve the performance of the more important Class A frames in an audio-video-bridging (AVB) switch, which adopts a credit-based shaper mechanism. We propose a constant bandwidth server to replace the credit-based shaper while taking fairness into consideration. Evaluation with extensive experiments shows that both the resilience and efficiency of the TSN are significantly enhanced. For the delayed ST frames and event-triggered traffic, our approach can schedule more than the solution using the AVB switch, even with a high network load.

5.1 Introduction

Advances in the automotive industry in recent decades have led to a dramatic increase in the number of sensors and electronic control units (ECUs) in vehicles to support new functions, including advanced driving assistance systems (ADAS) and automated driving systems (ADS). As a consequence, there is more time- and safety-critical communication within the vehicle for which guarantees of bounded latency and low jitter are required. This inevitable trend has forced the industry to research next-generation *in-vehicle networks* (IVNs), as conventional IVNs do not have the bandwidth capability to support such a large amount of data traffic.

The traffic in such systems, especially safety-critical data, should be transmitted with hard real-time guarantees and with no or minimal interference. Therefore, broadband IVNs with high reliability, deterministic behaviour and stringent traffic isolation have become necessary — indeed, vital — for guaranteeing the system's performance and safety. *Time-Sensitive Networks* (TSNs) enable the use of standard Ethernet as a real-time communication network and allow network sharing among multiple time-critical applications, and non-time-critical applications [111], [57]. The IVN structure we consider in this work comprises a central TSN switch that is connected to multiple zones or domains (e.g., for handling LiDARs) in the vehicle and acts as a backbone. Each domain connects to the TSN backbone through a domain control unit (DCU) for cross-domain data exchange [128], [59], [121], [113]. Figure 5.1 illustrates this network architecture.

The traffic in the IVN can be divided into three classes reflecting their importance. Messages directly related to safety are recognised as *safety-critical traffic* (ST), whereas the flows carrying information with lower importance but which can provide mission-critical services are generally classified as *Class A*. Other traffic with even lower importance can be regarded as *Class B*. The gate control list (GCL) is recognised as the schedule, calculated offline under the assumption that all safety-critical traffic frames can be released and treated "safely", ideally as predefined. The schedule is fixed in the switch at run-time to guarantee the deterministic transmission of safety-critical periodic traffic and realise time isolation to avoid interference among different frames [36]. In practice, the unexpected delay of safety-critical frames, which is inevitable due to system complexity and the unpredictable operating environment, injects uncertainty into the network. Several factors contribute to this phenomenon.

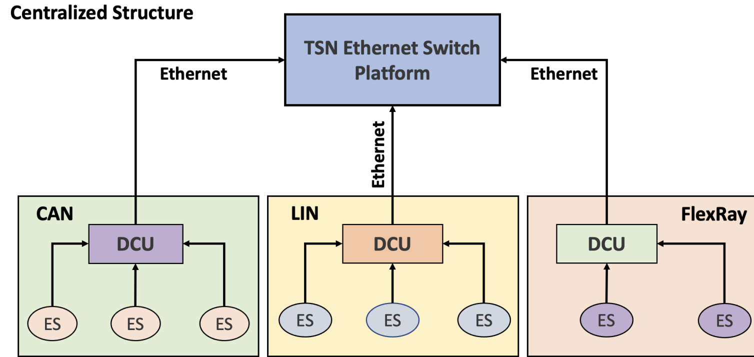

Figure 5.1: TSN-based in-vehicle network architecture

As introduced in the previous paragraph, various domains are interconnected through the TSN backbone. Jitter originating from any of these domains can result in varying arrival times during the runtime of ST frames released from specific domains. Such variation can be attributed to disparities in routing paths within each domain, differing processing times at devices, and other relevant factors. Additionally, within each domain, the occurrence of packet drops due to network errors or congestion can disrupt the expected periodic arrival pattern. The need to retransmit lost packets can introduce extra delays and irregularities. Moreover, dynamic alterations in the domain network topology, such as link failures or route recalculations, can impact packet delivery, consequently leading to fluctuations in arrival times [11]. In this work, our focus is on the TSN backbone rather than addressing issues specific to each individual domain. Without any further measures, once the frame misses its transmission time slot allocated by the GCL, the timing behaviour of the ST queue (FIFO) is undermined, and the delayed frame may disturb the order of other messages and cause a severe knock-on effect. In addition to the unexpected arrival of ST frames, some event-triggered traffic may have relatively high priority. With a standard AVB switch, such traffic is fed into the Class A queue. We need to transmit this traffic quickly and raise the throughput to improve the quality of service (QoS). Although the use of event-triggered traffic is intended to improve system safety and reliability, a fixed schedule cannot handle the uncertainties, dramatically reducing the flexibility of the TSN-based system.

According to common practice, frames with incorrect arrival times are dropped directly to prevent the adverse effect from propagating. Most current

work focuses on dealing with permanent faults caused by physical link failure or temporary faults caused by packet loss. The lack of research on mechanisms to deal with temporal faults caused by incorrect frame arrival impedes the application of TSN-enabled systems.

In this work, we propose a new scheduling strategy, namely *Resilient and Efficient Time-Sensitive Networking (reTSN)* to address the aforementioned issues. To illustrate the superiority of our proposed method, we chose the state-of-the-art credit-based shaper mechanism-based AVB TSN as the baseline approach. The evaluation results demonstrate that our proposed method can substantially decrease the drop-outs of safety frames across various workloads. Furthermore, our method prioritizes the transmission of more crucial Class A frames while maintaining fairness for less significant Class B frames.

The **main contributions** can be summarised as follows:

- reTSN is a novel strategy adopting a fully multi-level preemption concept to realise dynamic priority allocation, which can reduce the waste of bandwidth and dramatically improve the scheduling effectiveness of TSN.

- reTSN introduces the notion of an emergency queue technique, which integrates acceptance-test and run-time priority allocation strategy to deal with the delayed ST frames and can schedule up to four times as many delayed ST frames than the AVB-based solution, even in a relatively highly loaded network (e.g., with 80% utilisation).

- For unanticipated traffic, we propose a constant bandwidth server-based strategy to replace the conventional credit-based shaper mechanism to reduce the latency of more important Class A frames. This strategy also takes the fairness of Class B frame transmission into consideration. Under a relatively highly loaded network, the benefit is more noticeable.

The rest of the chapter is organised as follows: In Section 5.2, the proposed reTSN method is detailed, followed by the formulation of proposed strategies for delayed ST and event-triggered traffic in Section 5.3. The evaluation is introduced in Section 5.4. Finally, in Section 5.5, we conclude this work with a discussion of future work.

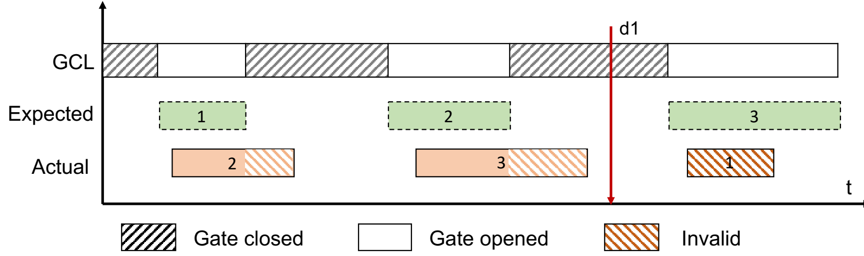

Figure 5.2: Temporal fault caused by a delayed traffic frame

5.2 Method Overview

A network's resilience and efficiency determine its capability to handle uncertain traffic conditions. In this work, countermeasures to uncertainty are presented from two perspectives:

1. The delayed ST frame is recognised as a temporal fault. The proposed method adopts a slack-stealing algorithm, emergency queue and multi-level preemption mechanism to reduce the likelihood that a frame is discarded and improve the capability for temporal fault tolerance; and

2. The event-triggered traffic with unknown release time, which should be transmitted as quickly as possible. Cooperating with a multi-level preemption mechanism and constant bandwidth server, the virtual deadline for event-triggered frames with higher priority (Class A) can be allocated dynamically to reduce transmission latency.

In the subsequent subsections, motivational examples are given, and the reTSN design is described comprehensively.

5.2.1 Motivational examples

Motivational example 1

Figure 5.2 depicts the fault caused by a delayed ST frame. Assume all frames arrive at the switch sequentially and are scheduled according to the predefined time slots (via GCL). Ideally, frame 1 should arrive before frame 2. If frame 1 is delayed and arrives at the switch after frame 3, the order will be disturbed. If frame 2 is inserted into the ST queue before the closing time point of the frame 1 slot, it will be transmitted immediately. If the remaining frame 1 time slot is too short to complete frame 2 transmission, that may result in one

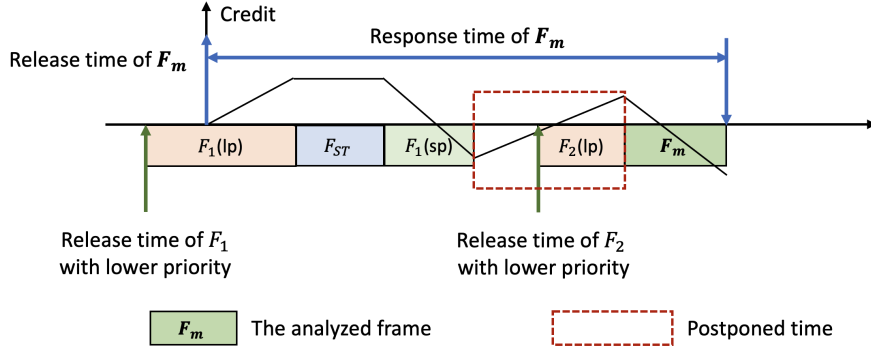

Figure 5.3: Frame transmission using a credit-based shaper [25]

frame becoming invalid. Frame 3 can be invalidated the same way. For the delayed frame 1, though it can be transmitted within the time slot belonging to frame 3, it is inevitable that it will miss its deadline. According to the current mechanism, it will be discarded if a frame is invalid or doesn't arrive at the correct time point. However, dropping a safety-critical frame (and potentially a sequence of safety-critical frames) could affect system safety and cannot eliminate the effect caused by the timing fault. Therefore, instead of dropping the frame directly, further effort should be made to finish transmitting delayed frames before their deadlines to reduce drop-outs with the potential to impact safety. This also suggests that the GCL must be dynamically modified to avoid incorrect time slot occupation. The method proposed in this work can improve the fault tolerance capability of the TSN switch from these two perspectives.

Motivational example 2

To deal with unpredictable event-triggered traffic, existing reservation mechanisms can prevent bursts in transmission based on the credit-based shaper algorithm introduced in [25]. The pending frames can be transmitted if and only if the associated credit is non-negative. During its transmission, the credit is reduced at a constant rate called *sendSlope* but frozen during the transmission of ST frames. When the frames are pending in the queue, the credit increases at a constant rate known as *idleSlope*. The credits are initialised to zero, and the parameter definition of *sendSlope* and *idleSlope* can be considered as a fine-tuned procedure for obtaining optimal traffic transmission performance. In different scenarios, the adjustment procedure can be time-consuming. Figure 5.3 illustrates one of the representative cases of

credit-based shaper strategy-based frame transmission with non-preemption. Assume frame F_m is being analysed, and the line is its credit. According to the aforementioned rule, its transmission can be blocked by safety-critical traffic F_{ST}. Moreover, the frames $F(sp)$ with the same priority (sp), which are pending before it in the same queue, and even $F_1(lp)$, with a lower priority (lp) but with an earlier release time, can prevent the frame's transmission. Furthermore, if the credit is not larger than zero, the frame is not allowed to be transmitted, even when the channel is idle. Before the credit is reset to zero, suppose that a second frame with lower priority $F_2(lp)$ is released. In that case, the actual start time of F_m must be postponed. The red dash box outlines the postponed time of F_m, which cannot be eliminated by fine-tuning the parameters [25].

5.2.2 System model

The design of the model is based on the assumption that safety-critical periodic traffic is scheduled by GCLs and that the buffer size for the queues is not unlimited but sufficient (this can be determined statically, off-line).

Network model

ST traffic is released periodically and put into the ST queue. We assume the off-line calculated schedule (GCL) is frame-level and repeated cyclically. The event-triggered traffic is divided into two classes according to their minimum inter-arrival time (highest release frequency) at a pre-specified threshold; this can be determined *a priori*. In this work, the priority of each frame is determined according to its deadline, and the class A queue is used for frames with an inter-arrival time less than the threshold, which also implies queuing the frame with higher priority. The Class B queue is used for traffic with an inter-arrival time greater than the threshold. For example, frames with an inter-arrival time less than 100 are fed into the high-priority Class A queue. In contrast, those with inter-arrival times greater than 100 are inserted into the Class B queue. Although there is no hard real-time requirement for Class A and Class B frames, we must provide faster transmission to increase the throughput, especially for Class A frames.

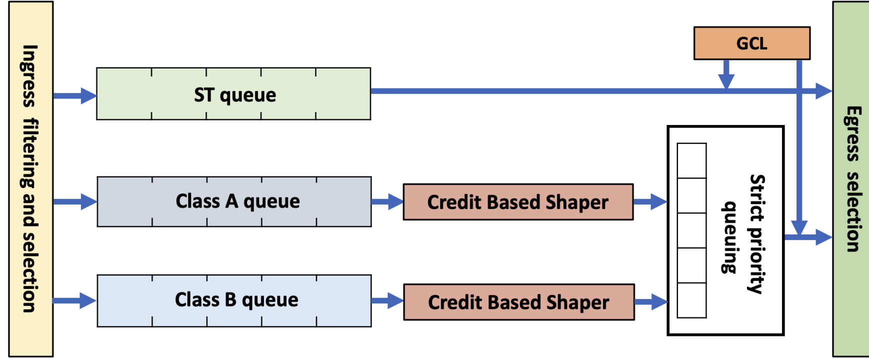

Figure 5.4: The conventional AVB TSN switch

Traffic model

In this work, each flow consists of a sequence of frames. The routes for all flows are predefined at design time and are deterministic at run-time. The ith frame of flow F_j can be characterized as:

$$f_i^j := \{C_i^j, T_j, D_i^j, r_i^j\} \tag{5.1}$$

where C_i^j represents the frame size, and T_j and D_i^j denote the frame period and relative deadline, respectively. For event-triggered frames, T_j and D_i^j are defined by the minimum inter-arrival time. r_i^j represents the release time of the ith frame of the jth flow. Moreover, we assume that $T_j = D_i^j$.

5.2.3 Conventional AVB model and credit definition method

Figure 5.4 demonstrates the conventional AVB switch. The priority of an individual frame is defined according to the most recent credit of the corresponding queue. According to the definition of the credit-based shaping mechanism [126], the pending frames can be transmitted only if the associated credit is non-negative. Additionally, during the transmission, the credit is reduced at a constant rate called *sendSlope*. When the frames are pending in the queue and waiting for transmission, the credit increases at a constant rate known as *idleSlope*. Moreover, if there is no pending frame and the credit of the corresponding class is negative, the credit also accumulates at the *idleSlope* rate until it reaches zero. The credits are initiated at zero. Equations 5.2 and 5.3 illustrate how the *sendSlope* and *idleSlope* can be calculated [67]. *portTransmitRate* refers to the maximum data transfer rate that a specific

network port can support. It uses the same units as bandwidth. The *idleSlope* represents the bandwidth reserved for the queue by the bridge, while the *sendSlope* corresponds to the transmission rate of the port MAC service.

$$sendSlope = idleSlope - portTransmitRate \tag{5.2}$$

$$idleSlope = reservedBytes/FrameIntervalTime \tag{5.3}$$

Although the credit-based shaper based method prevents traffic bursts, the buffering delay caused by the increasing credit cannot be neglected. We denote the *idleSlope* as iS_a for Class A and iS_b for Class B. The values are defined according to the remaining bandwidth after allowing for ST periodic traffic (i.e. $U_{AVB} = 1 - U_{ST}$). The *sendSlope* for Class A and Class B are denoted by sS_a and sS_b respectively. In this work, the values of *idleSlope* and *sendSlope* for Class A and Class B are assumed to be the same thus no bias is introduced and once a frame from any class gets the transmission opportunity, all the remaining bandwidth, after accounting for ST periodic traffic, can be used to support its transmission. Fine-tuning of these parameters subject to the application could potentially further improve the performance but analysing this is beyond the scope of this work. As the motivational example demonstrates, tuning the parameters does not solve the underlying fundamental problem. Based on Equations 5.2 and 5.3, the slopes used in this work can be calculated as follows:

$$\begin{cases} iS_a = iS_b = U_{AVB} \\ sS_a = sS_b = U_{AVB} - 1 \end{cases} \tag{5.4}$$

The conventional AVB model adopts a one-level preemption mechanism. Only ST frames can preempt the transmission of non-ST frames (Class A and Class B). Class A and Class B frames are transmitted based on strict priority, which is defined by the credit-based shaper; Class A and Class B frames cannot preempt each other.

5.2.4 reTSN model

Figure 5.5 illustrates our proposed reTSN switch. Compared to the AVB switch, we adopt *dynamic GCL modification*. If the current ready frame is not the expected frame, the gate to the ST queue will be closed, and the gates to other queues will be opened. The necessity of multi-level preemption began

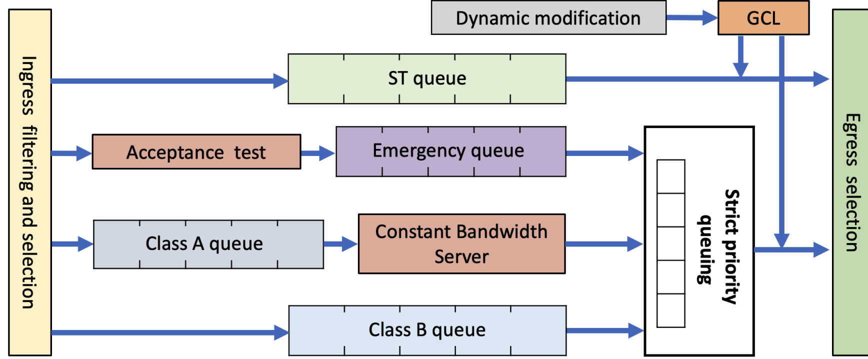

Figure 5.5: The constant bandwidth server based TSN switch

to be emphasised in recent studies such as [99] and [10]. However, the existing work only enables the preemption of Class B by Class A (Class A can not be preempted by Class B) while still adopting the credit-based mechanism. The unpromising results caused by credit recharging (increasing credit at *idleSlope*) remain unsolved. Thus, this work relaxes the limitation of the one-level preemption mechanism and adopts a *multi-level preemption* mechanism, which enables all frames to preempt or be preempted according to their latest allocated priorities. This operational mechanism bears resemblance to execution preemption, which is commonly applied in task scheduling on computational platforms. In recent years, the authors of the paper [99] were the first to provide an in-depth explanation of how to implement a multi-level preemption scheme in TSN. In our work, we leverage the concept of multi-level preemption to address the limitations observed in existing methods. Frame preemption involves two MAC services for an egress port: preemptable MAC (pMAC) and express MAC (eMAC). Express frames can interrupt preemptable frame transmission. Upon resuming, the MAC merge sublayer reassembles frame fragments at the next bridge. Preemption incurs computational overhead in the link interface due to transitioning to express frames. A frame transmission pauses, and the incomplete frame gets a cyclic redundancy check (CRC). When high-priority traffic has been sent, interrupted transmission resumes in the next time slice.

For safety-critical periodic traffic, the schedule is calculated offline based on the *earliest deadline (absolute-deadline) first* (EDF) algorithm and repeats according to the hyperperiod T_{hp} of all safety-critical flows. We assume that the ST frames are ideally released according to predefined period. To limit

the complexity of this work, each frame's actual release time can be regarded as the arrival time at the switch, which can be measured; thus, the absolute deadline can be calculated following Equation (5.5), where ϕ_j represents the release time of the first frame instance ($\phi_j = r_1^j$) of traffic flow F_j and D_i^j denotes the relative deadline. However, this assumption is potentially invalid for some traffics at run-time because of the inevitable jitter. In that case, the unexpected arriving frames can be treated as delayed frames by following introduced emergency queue-related operations.

$$d_i^j = \phi_j + (i-1)T_j + D_i^j \tag{5.5}$$

Suppose an ST frame is delayed. Instead of dropping it directly, the proposed acceptance test will be invoked to identify whether the slack is sufficient for the frame's transmission. If so, the delayed ST frame will be fed into the emergency queue and assigned with a small virtual absolute deadline (e.g. zero). In that case, it is guaranteed to have the smallest value compared with the absolute deadlines of other non-ST frames and is allowed to preempt all non-safety-critical traffic with the highest priority. The detailed formulation of the acceptance test is further detailed in Section 5.3.1. Different to the urgency queue introduced in paper [97], which follows a token bucket algorithm to regulate the traffic and is similar to the CBS-based strategy, queuing delay and the postponed time caused by token recharging can not be eliminated. In our work, the acceptance test based enqueue and highest priority allocation operation can guarantee the immediate transmission of delayed ST frames. Furthermore, the waste of bandwidth consumed by some delayed ST frames, whose deadline missing cannot be avoided, can be reduced.

For event-triggered traffic, we assume that the physical transmission time from the source to the switch can be neglected. The actual release time of each frame r_i can be regarded as the arrival time at the switch, and the absolute deadline d_i can be calculated by $d_i = r_i + D_i$, where D_i denotes the predefined relative deadline. For Class A-frames, the constant bandwidth server is utilised to determine the virtual relative deadline dynamically. The method of server definition is defined in Section 5.3.2. For Class B frames, the absolute deadlines are calculated based on their release time and relative deadlines (minimum inter-arrival time). Once the absolute deadlines of all ready frames are defined, they are scheduled following strict priority order. With the adoption of the multi-level preemption mechanism, Class A frames are able to preempt Class B frames.

As shown in Figure 5.6, the time-aware scheduler plays a crucial role in preventing interference or overlap between frames carrying safety-critical traffic. Its objective is to ensure that these frames remain undisturbed, upholding data integrity. To achieve this, the time-aware scheduler introduces a guard band (purple box) before each time slice dedicated to ST traffic (blue-grey box). During this guard band period, new Ethernet frame transmissions are prohibited, allowing only ongoing transmissions to conclude [115]. Despite their effectiveness in safeguarding high-priority critical traffic, guard bands entail a significant drawback. The time allocated to a guard band is essentially lost; it cannot be harnessed for new data transmission. Consequently, the time sacrificed to guard bands directly translates into reduced bandwidth available for background traffic on the specific Ethernet link. Without a preemption mechanism, the duration of this guard band must match the time it takes to safely transmit the maximum frame size. For an Ethernet frame compliant with IEEE 802.3, including a single IEEE 802.1Q VLAN tag and accounting for inter-frame spacing, the total length is calculated as follows: 1500 bytes (frame payload) + 18 bytes (Ethernet addresses, EtherType, and CRC) + 4 bytes (VLAN Tag) + 12 bytes (Interframe spacing) + 8 bytes (preamble and SFD) = 1542 bytes. The total time required to send this frame varies according to the link speed of the Ethernet network. In the case of Fast Ethernet with a transmission rate of 100 Mbit/s, the guard band needs to be at least 123.36 µs long. Frame preemption minimizes guard band size, determined by the preemption mechanism's precision. This is the reason why we introduce a preemption strategy in this work – to reduce the bandwidth waste caused by guard bands.

In this work, a minimum valid transmission time C_{vef} is assumed to be one time unit. The minimum preemptable frame size is set to 128 bytes with a two time unit transmission time C_{minp}. The guard band size is set to 128 bytes, and once a preemption from an ST frame occurs, the remaining transmission time of the current transmitted non-ST frame is checked. With the adoption of a *without hold and release preemption mechanism* [25] if the remaining time C_{remain} is more than C_{minp}, the frame is allowed to continue transmitting for one time unit to reduce the wasted bandwidth caused by the guard band. After that, the preemption overhead (including a header and cyclic redundancy check (CRC)) is appended to the remaining part to create a new frame. If the remaining time is less than or equal to C_{minp}, the frame

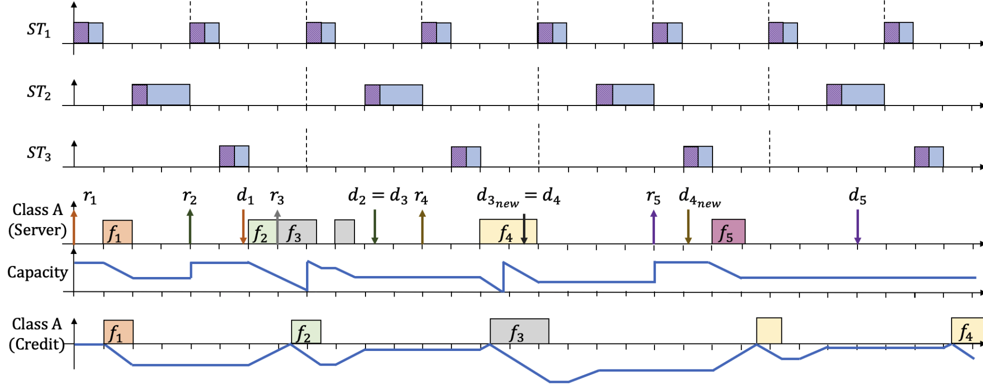

Figure 5.6: Example of multi Class A frames transmission ($T_{ser} = 23$, $C_{ser} = 8$, $iS_a = 0.375$, $sS_a = -0.625$)

actively undergoing transmission can finish its transmission. Equation (5.6) describes the continuous transmission time C_{conti} of the current transmitted frame after preemption takes place.

$$C_{conti} = \begin{cases} C_{vef} & \text{if } C_{remain} > C_{minp} \\ C_{remain} & \text{if } C_{remain} \leq C_{minp} \end{cases} \tag{5.6}$$

5.2.5 Transmission performance comparison

In order to intuitively demonstrate the difference in frame transmission between the server-based method and the AVB-based method, Figure 5.6 illustrates the case of Class A traffic transmission. Purple boxes represent guard bands, and blue boxes depict the time slots for frame transmission. This case study compares the transmission performance based on credit-based shaper mechanisms and the proposed constant bandwidth server-based strategy. In this figure, r_i represents the ready time of the Class A frame instance, d_i denotes the server-allocated deadlines, and d_{inew} is the updated server deadline in light of capacity exhaustion or the remaining capacity available to satisfy the requirement described in Section 5.3.2. In the server-based method, the transmission can start immediately when the channel is idle. Therefore, the response time of event-triggered frames is decreased, and within the observation time duration, more frames can be transmitted using the proposed method. For frame f_2, the guard band allows it to continue to transmit for one time unit without being preempted. Frame f_5 in this example can finish its transmission before the corresponding server deadline. However, the credit-based

shaper strategy cannot transmit this frame within the time duration. Although the credit-based shaper method can prevent traffic bursts, the buffering delay caused by credit recharging cannot be neglected, as explained by motivational example 2. Please note that the primary focus of this figure is to intuitively highlight the advantages of employing a server. These advantages include the ability for the capacity budget to immediately return to its maximum value upon frame release and the dynamic adjustment of frame priorities based on dynamically allocated deadlines. This is achieved without encountering delays due to the credit recharging rule. Therefore, specific values for the server budget and credit values at each time point are not necessarily included in the figure.

5.3 reTSN Formulation

In this section, the proposed strategy is formulated separately for (a) resilience in terms of the improvement of temporal fault tolerance and (b) efficiency of event-based traffic.

5.3.1 Delayed Safety-critical Traffic

As mentioned in Section 5.2.4, an emergency queue is adopted to handle delayed traffic. The acceptance test must analyse the idle time according to the offline calculated schedule (GCL) and the current ongoing transmitted frame to decide whether a delayed ST frame can still be accepted or whether it must be discarded directly.

For the acceptance test, the slack-stealing algorithm is adopted, which enables the delayed frame to be transmitted before its deadline by utilising all possible idle time from ST periodic traffic without causing other frames to miss deadlines. For example, suppose the ith ST frame of the jth flow is delayed with the actual arrival time r'^{j}_{i}. Only if it passes the slack analysis-based acceptance test can its transmission be guaranteed, meaning it will arrive at the destination within its deadline denoted by d'^{j}_{i}. The slack in the time duration from the actual arrival time of the delayed ST frame to its deadline can be analysed following Equation (5.7).

$$slack(t) = d'^{j}_{i} - r'^{j}_{i} - \mathcal{I}_{ST} \tag{5.7}$$

According to the GCL, the start time t^{k}_{hS} and end time t^{k}_{hE} of each frame

instance are known (h and k represent the frame and flow indices, respectively), and the interference $\mathcal{I}_{ST}$ caused by ST scheduled traffic arises from three aspects:

$$\mathcal{I}_{ST} = \mathcal{I}_{act} + \mathcal{I}_{fut_1} + \mathcal{I}_{fut_2} \tag{5.8}$$

1. $\mathcal{I}_{act}$: interference from the active scheduled ST time slot with $t^k_{hS} \le r'^j_i$ and $r'^j_i < t^k_{hE} < d'^j_i$ (Please note that we don't need to consider $t^k_{hS} \le r'i^j$ and $t^k_{hE} \ge d'^j_i$ because that would indicate the absence of any slack time within the time duration from the actual arrival time of the delayed ST frame to its deadline.):

$$\mathcal{I}_{act} = \sum_{\forall i \in (t^k_{hS} \le r'^j_i \bigwedge r'^j_i < t^k_{hE} < d'^j_i)} (t^k_{hE} - r'^j_i) \tag{5.9}$$

2. $\mathcal{I}_{fut_1}$: interference from the future scheduled ST time slot with $t^k_{hS} \ge r'^j_i$ and $t^k_{hE} \le d'^j_i$:

$$\mathcal{I}_{fut_1} = \sum_{\forall i \in (t^k_{hS} \ge r'^j_i \bigwedge t^k_{hE} \le d'^j_i)} (t^k_{hE} - t^k_{hS}) \tag{5.10}$$

3. $\mathcal{I}_{fut_2}$: interference from the future scheduled ST time slot, but with $r'^j_i \le t^k_{hS} \le d'^j_i$ and $t^k_{hE} > d'^j_i$:

$$\mathcal{I}_{fut_2} = \sum_{\forall i \in (r'^j_i \le t^k_{hS} \le d'^j_i \bigwedge t^k_{hE} > d'^j_i} (d'^j_i - t^k_{hS}) \tag{5.11}$$

It is worth noting that the time slot allocated to the transmission of the ST frame C^j_i consists of the transmission time of ST frame and guard band C_{gb}. If the frame is delayed, the actual transmission time excludes the guard band part and is denoted as C'^j_i. (i.e., $C^j_i = C'^j_i + C_{gb}$)

Based on Equation (5.12), the maximum preemption times and the related maximum preemption overhead ξ_i can be calculated where ω denotes the overhead per preemption. As introduced earlier, we assume a minimum valid transmission time of one time unit. The minimum preemptable frame size is set to 128 bytes with a two time unit transmission time. To ensure successful preemption, the minimum remaining frame size must allow either preemption at the start of the guard band (where the remaining frame can still be valid) or completion of its transmission within the guard band. Considering these conditions, we can determine the maximum number of times a frame

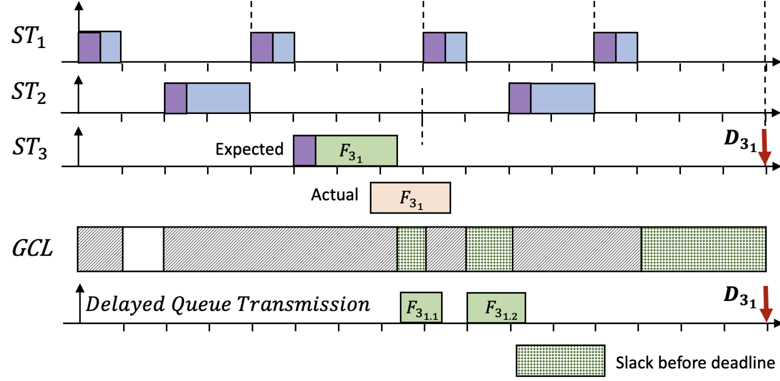

Figure 5.7: Example of Delayed ST frame transmission

can be preempted, given by $\lceil C'^{j}_{i}/2 \rceil$, where 2 represents the minimum preemptable frame size indicated by the time it consumes. Subsequently, we can calculate the overhead and update the transmission time while accounting for the additional overhead. The updated transmission time, which includes its preemption overhead, can be estimated using Equation (5.13). The maximum blocking time from the frame with lower priority is set to C_{vef}.

$$\xi_i = \lceil C'^{j}_{i}/2 \rceil * \omega \tag{5.12}$$

$$C'^{j}_{i} \leftarrow C'^{j}_{i} + \xi_i \tag{5.13}$$

$$slack(t) \geq C'^{j}_{i} + C_{vef} \tag{5.14}$$

The delayed ST frame can be accepted and fed into the emergency queue if the slack time fulfils in-equation (5.14). Otherwise, the deadline missing warning and rejection operation will be reported to the application layer to trigger safety-related activities. The time complexity of the acceptance test is $O(N)$, where N is the number of GCL scheduled ST frames within one hyperperiod.

Figure 5.7 illustrates an example of delayed frame transmission. Frame F_{3_1} is the analysed frame which did not arrive at the switch as expected. However, based on the slack time calculation following Equation (5.7), under the current circumstances, there are sufficient spare time slots (marked by green prismatic grids) to guarantee the transmission of the delayed ST frame before its deadline D_{3_1}. That means the delayed frame can pass the acceptance test and be fed into the emergency queue. Once the gate of the ST queue is

closed, the delayed frame can be transmitted immediately with the highest priority.

5.3.2 Event-triggered Traffic

The release time of event-triggered traffic is unpredictable and is a source of uncertainty for the system. Event-triggered traffic is divided into two: Class A and Class B, according to their minimum inter-arrival times. A constant bandwidth server is used to determine the virtual relative deadlines of the Class A frames. The relative deadlines of the Class B frames are defined by their minimum inter-arrival times. Their absolute deadlines, used to determine their priorities, can be calculated according to their actual release time and the virtual relative deadline, as shown in Section III-D.

Constant bandwidth server-based deadline allocation for Class A frames: The server-based deadline allocation can be understood as follows: when a frame is ready to be transmitted, an appropriate deadline is allocated, then the priority of the frame is determined, and it may preempt or be preempted according to the deadline. If the transmission time of the ongoing transmitted frame would exceed the expected time defined by the server, its deadline is postponed, meaning its priority is decreased, and another frame could receive the opportunity to be transmitted. This property prevents channel occupation by frames from the same class. Because of the postponed deadline, the priority of the frame with a closer deadline from another queue may be increased. The detailed procedure of a constant-bandwidth server-based deadline allocation for Class A frames can be described as follows [35]:

- The utilisation allocated to the server U_{ser} is C_{ser}/T_{ser}. The deadline assigned by the server d_{ser} is initialised to zero. C_{ser} represents the maximum expected transmission time and T_{ser} denotes the server's period;

- A virtual deadline d_i is dynamically allocated to the frame located at the head of the Class A queue following $d_i = d_{ser}$, and whenever the frame is transmitted, the capacity is decreased by the same amount as the transmission time;

- When the maximum expected transmission time (capacity) is exhausted, the capacity is replenished to the maximum value and the server deadline is updated as $d_{ser} = d_{ser} + T_{ser}$;

- When a frame is ready at time point r_i, and the remaining capacity $C_{remain} \geq (d_{ser} - r_i) * U_{ser}$, a new deadline is generated $d_{ser} = r_i + T_{ser}$, and the capacity is recharged to the maximum value. Otherwise, the frame is transmitted with the last server deadline and consumes the remaining capacity.

Constant bandwidth server parameters definition: The constant bandwidth server can be considered as a source of virtual periodic traffic which can be scheduled with other periodic traffic, using the EDF strategy.

According to the definition of the EDF algorithm, Equation (5.15) must be fulfilled to guarantee the schedulability of the network. The upper bound of utilisation allocated to a constant bandwidth server can be defined by Equation (5.16), and the utilisation allocated to a constant bandwidth server equals the upper bound. U_p represents the utilisation of all periodically released ST traffic. Please note that in this work, the deadlines for non-safety-critical tasks are not necessarily guaranteed. The function of the server is to enable these tasks to begin transmission without any delay caused by credit recharging. The bandwidth upper bound for the server U_{ser}, is utilized to establish server parameters and provide dynamic deadlines for Class A. This approach prevents prolonged channel occupancy, which could otherwise hinder the transmission of Class B frames.

$$U_{ser} + U_p \leq 1 \tag{5.15}$$

$$U_{ser} = U_{sup} = 1 - U_p \tag{5.16}$$

Under the worst-case scenario, the frames belonging to other categories can interfere with the transmission. If the transmission time of the hth Class A frame C_h is not a multiple of the server capacity C_{ser}, the remaining portion of the virtual frame will end at most Δ_h time units before the reassigned server deadline.

$$\Delta_h = \left\lceil \frac{C_h}{C_{ser}} \right\rceil C_{ser} - C_h \tag{5.17}$$

The response time can be calculated as follows:

$$R_h = \frac{C_h}{C_{ser}} T_{ser} - \Delta_h = C_h + \left\lceil \frac{C_h}{C_{ser}} \right\rceil (T_{ser} - C_{ser}) \tag{5.18}$$

Once the capacity is exhausted, the server's deadline is postponed, and other frames can preempt the served frame because of their smaller deadlines.

The preemption overhead can be taken into account by subtracting it from the capacity, and the equation for the response time calculation is updated using Equation (5.19). According to Equation (5.30), the maximum preemption overhead of the server is denoted as Ω_{ser}.

$$\begin{aligned} R_h &= C_h + \left\lceil \frac{C_h}{C_{ser} - \Omega_{ser}} \right\rceil (T_{ser} - C_{ser} + \Omega_{ser}) \\ &= C_h + \left\lceil \frac{C_h}{T_{ser}U_{ser} - \Omega_{ser}} \right\rceil (T_{ser} - T_{ser}U_{ser} + \Omega_{ser}) \end{aligned} \tag{5.19}$$

The definition of server parameters (C_{ser} and T_{ser}) can be formulated as a minimization of the upper bound of response time R_h. The upper bound of the response time can be determined by the following function, which is proven in [35].

$$R_h^{up} = T_{ser} - C_{ser} + \Omega_{ser} + \frac{T_{ser}}{C_{ser} - \Omega_{ser}} C_h \tag{5.20}$$

Based on Equation (5.20) and Equation (5.16), the average response time $\overline{R_h}$ can be defined by Equation (5.21), where $\overline{C_h}$ denotes the average transmission time of frames from Class A.

$$\overline{R_h} = T_{ser}(1 - U_{ser}) + \Omega_{ser} + \frac{T_{ser}}{T_{ser}U_{ser} - \Omega_{ser}} \overline{C_h} \tag{5.21}$$

Equation (5.22) is the partial derivative of the average response time $\overline{R_h}$ with respect to the server period T_{ser}. The minimization of the average response time can be simplified by functional analysis.

$$\frac{\mathrm{d}\overline{R_h}}{\mathrm{d}T_{ser}} = 1 - U_{ser} - \frac{\Omega_{ser}}{(T_{ser}U_{ser} - \Omega_{ser})^2} \overline{C_h} \tag{5.22}$$

By setting (the value of) Equation (5.22) to zero, the appropriate server period can be defined using Equation (5.23) and, based on Equation (5.16), the server capacity can be calculated using Equation (5.24).

$$T_{ser} = \left\lfloor \frac{1}{U_{ser}} (\Omega_{ser} + \sqrt{\frac{\Omega_{ser}\overline{C_h}}{1 - U_{ser}}}) \right\rfloor \tag{5.23}$$

$$C_{ser} = \left\lfloor T_{ser}U_{ser} \right\rfloor \tag{5.24}$$

After determining the parameters of the constant-bandwidth server, the priority of the frame at the head of the Class A queue can be allocated dynamically

according to the virtual relative deadline defined by server $D_i = T_{server}$. As aforementioned, the absolute deadline of the ready Class A frame can be calculated as $d_i = r_i + D_i$. According to the EDF strategy, the priorities of event-triggered traffic can be defined and then scheduled based on strict priority.

5.3.3 Response Time Analysis

In this work, we adopt a cooperative scheduling strategy. The worst-case response time of accepted delayed ST frames and event-triggered traffic frames can be analysed following the same method and is denoted R_i. The response time can be calculated according to Equation (5.25).

$$R_i = C_i - C_{conti} + L(r_i, R_i) + \Omega + B_i \tag{5.25}$$

where C_i is the transmission time of the analysed frame. Please note that the guard band time is included in the transmission time of the ST frame during the interference calculation. Without subtracting C_{conti}, there would be redundant calculations for the time overlapping with the guard band. r_i represents its release time and d_i denotes its absolute deadline. The transmission of the analysed frame can suffer interference from all the GCL-controlled ST frames during time interval $[r_i, R_i)$, and be blocked by an ongoing transmitted frame with lower priority B_i. As noted above, B_i is assumed as the minimum valid transmission time which equals one time unit. All activation times for ST frames can be regarded as critical instants, leading to worst-case interference to a lower priority frame. [25] and assume the ready time of the analysed frame is at a critical instant. Ω represents the overall preemption overhead. The iteration of Equation (5.25) terminates when the response time converges.

$L(r_i, R_i)$ depicts the interference in the time interval $[r_i, R_i)$. It can be divided into two terms, $R_a(r_i, R_i)$ and $R_f(r_i, R_i)$, where $R_a(r_i, R_i)$ is the interference caused by the currently active periodic ST frame f_{act} and event-triggered frames with deadline smaller than d_i, active time earlier than r_i and end time t_E after r_i; $R_f(r_i, R_i)$ is interference from the periodic ST frame instances coming in the future (activated after r_i) and event-triggered frames with deadline smaller than d_i.

$$L(r_i, R_i) = R_a(r_i, R_i) + R_f(r_i, R_i) \tag{5.26}$$

$$R_a(r_i, R_i) = \sum_{\substack{\forall k \in r_k \leq r_i < t_{kE}\ \& \\ \forall k \in ST\ \&\ \forall k \in hp_i(d_i)}} C_k \tag{5.27}$$

$$\begin{aligned} R_f(r_i, R_i) = &\sum_{\forall k \in ST} max(0, \left\lceil \frac{R_i - \zeta_k(r_i)}{T_k} \right\rceil) * C_k \\ &+ \sum_{\substack{\forall k \in r_i \leq r_k < R_i \\ \&\ \forall k \in hp_i(d_i)}} C_k \end{aligned} \tag{5.28}$$

As above, F_k represents the kth safety-critical flow and T_k is the corresponding period. $\zeta_k(r_i)$ describes the time instant greater than r_i at which the next periodic instance of ST flow F_k will be released and ϕ_k is the offset of ST flow F_k.

$$\zeta_k(r_i) = \left\lceil (r_i - \phi_k)/T_k \right\rceil T_k + \phi_k \tag{5.29}$$

The maximum preemption overhead can be calculated by Equation (5.30). The overall preemption overhead depends on the number of ST frames and event-triggered frames with higher priority released during the time interval $[r_i, R_i)$. ω denotes the overhead per preemption. N represents the number of event-triggered frames with deadline smaller than d_i and released during time interval $[r_i, R_i)$. Equation 5.32 can be utilized to ensure that the iteration can be stopped. If the calculation of R_i does not converge, the maximum number of preemptions and the maximum overhead term can still be used to determine the converged response time.

$$\xi_{max} = \lceil C_i/2 \rceil * \omega \tag{5.30}$$

$$\xi_i = \left(\sum_{\forall k \in ST} max(0, \left\lceil \frac{R_i - \zeta_k(r_i)}{T_k} \right\rceil) + N \right) * \omega \tag{5.31}$$

$$\Omega = min(\xi_{max}, \xi_i) \tag{5.32}$$

This formulation of the proposed reTSN enables its effectiveness to be evaluated by analysing the behaviour of differently loaded networks.

5.4 Evaluation

The evaluation is based on results produced using a simulation implemented in Python[1]. Existing simulation tools for the TSN network, e.g., *NeSTINg* [56], which are based on *OMNeT++*, are not used as they do not support multi-level preemption. UUnifast [53] is utilised to synthetically generate safety-critical periodic traffic (ST) to investigate the feasibility of the proposed method. The frames' relative deadlines are equal to their periods. Furthermore, EDF is adopted to calculate the schedule of ST frames offline; the schedule is then integrated into the switch as the GCL. The minimum valid frame size is chosen to be 64 bytes, and its transmission time is normalized as 1 time unit. We consider scenarios with different network loads. Within each scenario, there are ten safety-critical periodic and two event-triggered flows with randomly selected release times. The periods of ST frames are randomly selected from a pre-defined set $[50, 100, 200, 500, 1000]$, and the frame transmission time is derived from UUnifast-generated utilisation and related period. For Class A traffic, the minimum interval is set to 100 time units, and 200 time units is the minimum interval for Class B traffic. The transmission time is randomly selected between one and the mean value of periodic frames $[1, mean(\sum_{i=1}^{n} C_{STi})]$. The release time is randomly chosen between time instance $[0, 100]$. Moreover, the observation window (during which the frames' performance can represent the overall behaviour) is set to two hyperperiods which is enough to observe the main properties of traffic transmission with different strategies.

Table 5.1: The finish time variation of event-triggered frames without delayed ST frames

utilisation	Class	Improved (%)	Degraded (%)	Unchanged (%)	Mean reduced (%) (excluding zeros)	Median reduced (%) (excluding zeros)
0.3	A	17.89	0.04	82.07	-5.04	-1.39
	B	6.82	19.11	74.06	+0.88	+0.87
0.5	A	41.66	0.64	57.70	-5.86	-2.79
	B	11.85	42.52	45.63	+3.36	+1.49
0.6	A	62.23	1.55	36.22	-9.69	-5.63
	B	16.89	56.49	26.61	+4.61	+1.63
0.8	A	83.27	6.15	10.58	-23.14	-18.22
	B	48.07	44.39	7.54	-1.47	-0.27

[1]The code can be accessed at: `https://github.com/JIeSchnee/reTSN`

Table 5.2: The variation in the finish time of delayed ST frames

utilisation	Improved (%)	Degraded (%)	Unchanged (%)	Mean reduced (%) (excluding zeros)	Median reduced (%) (excluding zeros)
U_p=0.3	33.86	0.10	66.04	-2.73	-0.47
U_p=0.5	58.14	0.30	41.56	-3.70	-1.04
U_p=0.6	69.88	0.66	29.46	-6.06	-2.07
U_p=0.8	85.77	1.86	12.36	-10.77	-5.11

5.4.1 Performance without any delayed frame

As the first experiment, the performance of our proposed method is evaluated without any delayed frames. We synthesise task sets to simulate traffic under four different workloads ($U_p = 0.3$, $U_p = 0.5$, $U_p = 0.6$ and $U_p = 0.8$). The periodic flows regarded as safety-critical were generated randomly by a UUnifast-based traffic generator and the event-triggered traffic was generated as mentioned before. The simulation was used to run 10,000 trials for the four scenarios. Overall, there were around 300,000 event-triggered critical frames released within the observation window under each network load.

Consistent with the state-of-the-art approach proposed in [10], the result in Table 5.1 shows that our strategy can provide preference to Class A frames compared to the conventional methods, especially in a highly loaded network, where the response time of Class A frames is reduced by 23.14% on average. The transmission of class B frames was also guaranteed, though with performance degradation, which is acceptable because the values were less than 5% for the different workloads. Moreover, our approach can improve Class B frames' performance in a highly loaded network with $U_p = 0.8$. The simulation results showed that eliminating the credit-recharging procedure can undoubtedly benefit the non-ST frame transmission, which is consistent with the example of Figure 5.6.

5.4.2 Delayed ST frame

We assume that there is only one delayed ST frame during the observation time, and the frame instance is randomly selected from the offline schedule. The actual release time is randomly selected between its predefined release time and its deadline. Based on the understanding of the conventional AVB switch introduced in Section 5.2.3, the delayed ST frame can be considered

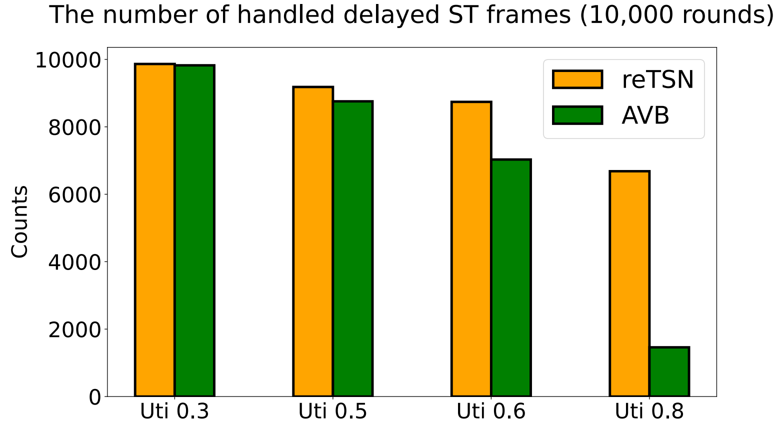

Figure 5.8: The number of delayed frames successfully handled by the proposed and conventional methods

the most important frame when compared to other non-ST frames. To avoid discarding it directly, the appropriate method is to insert it into the Class A queue. Moreover, according to the proposed method, once the delayed frame passes the acceptance test, it is inserted into the emergency queue. To compare the switch's capability to handle delay faults, we ran the simulation 10,000 times independently under four different network loads ($U_p = 0.8$, $U_p = 0.6$, $U_p = 0.5$ and $U_p = 0.3$). For each experiment trial, ST frames, event-triggered frames, and delayed ST frames were re-generated.

The number of schedulable delayed ST frames was collected under different network loads. Additionally, the actual finish time of the frames that were successfully handled by the two different methods were collected and recognized as comparable frames. As can be seen from Figure 5.8, the proposed switch can handle more delayed ST frames, especially when the network is heavily loaded. With $U_p = 0.8$, our approach can schedule four times more delayed ST frames than the AVB switch. By adopting an emergency queue, and allocating the highest priority to it, the probability that delayed ST frames will be treated immediately with minor interference is increased. Even in the lightly loaded network, the proposed strategy is advantageous and has a higher success rate.

The finish times of the proportion of delayed ST frames handled correctly by the proposed method and the AVB method were collected. The differences in the finish times were also recorded to evaluate the proposed method. The frames with the absolute difference of zero were ignored. D_R denotes the finish

time difference and is defined by Equation (5.33), where R_{AVB} represents the finish time based on the conventional AVB method, and R_{reTSN} denotes the finish time following the proposed constant-bandwidth server-based method. Furthermore, the ratio of the reduction in finish time to the finish time using the conventional method for each comparable frame was calculated according to Equation (5.34). A negative reduced value indicates an improved performance with a earlier finish time, and vice versa. These equations are as follows:

$$D_R = R_{reTSN} - R_{AVB} \tag{5.33}$$

$$P_R = [(R_{reTSN} - R_{AVB})/R_{AVB}] \times 100\% \tag{5.34}$$

Table 5.2 displays the finish time differences under four different network loads using two different methods. In a lightly loaded network, although the number of treated delayed ST frames was almost the same, 33.86% of the frames had earlier finish times, and they were reduced by 2.73% on average. When the network was under medium load, the number of delayed ST frames that can be handled by the conventional method dropped significantly. For the comparable frames, the proportion of improved cases increased considerably to 69.88%, when $U_p = 0.6$. In the highly loaded network with $U_p = 0.8$, over 60% of delayed ST frames were still able to successfully arrive at their destinations before deadlines using the proposed method. Further, 85.77% of the comparable frames had earlier finish times, and they were reduced by 10.77% on average, with the median reduction being 5.11%.

To summarize, the combined use of the emergency queue and the acceptance test enhances the switch's ability to tolerate temporal faults. Greater amounts of delayed ST traffic can be handled with a smaller finish time, especially under a highly loaded situation.

5.4.3 Event-triggered traffic

Before large-scale evaluation, we used randomly generated specific cases under four different workloads to illustrate the effect of the proposed method for each frame of the event-triggered traffic within the observation window. The frame-level comparison can intuitively demonstrate the source of the performance difference between our proposed method and the AVB approach. The traffic was generated using the same method as in subsection A.

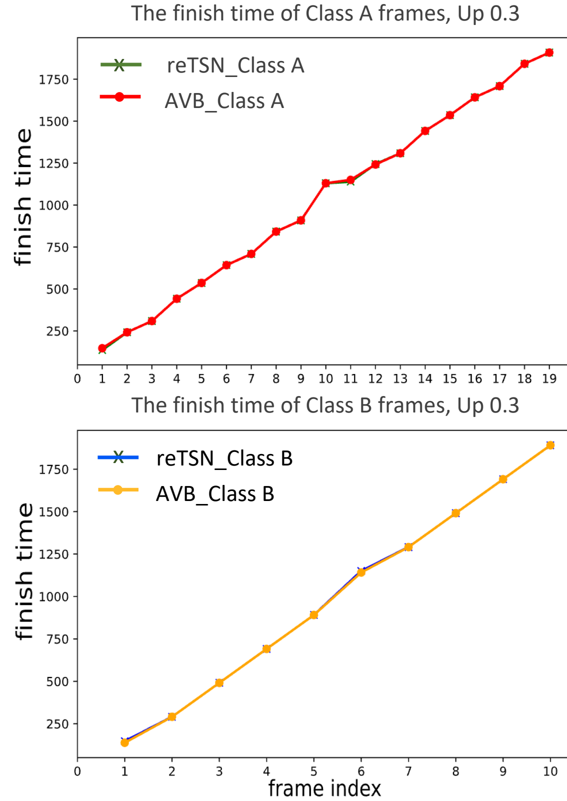

Figure 5.9: The finish time of event-triggered frames from Class A and Class B with $U_p = 0.3$ (index starts from 1)

First of all, the scenario with a low network load was observed, and the utilisation consumed by periodic safety-critical traffic was set to 0.3. The periods of randomly generated ST frames were [1000, 1000, 500, 1000, 200, 500, 500, 500, 200, 200], and the corresponding transmission times of the frame instances were set to [25, 9, 5, 33, 8, 6, 13, 3, 8, 17]. Thus, the exact utilisation consumed by periodic flows was $U_p = 0.286$. Further, the $29th$ frame instance in the GCL was selected as the delayed ST frame with an expected release time 1120, actual release time 1210 and deadline 1500. The delayed frame satisfies the requirement defined by the acceptance test. That means it can complete its transmission before the deadline and can be inserted into the emergency queue. For the event-triggered frames belonging to Class A, the release time of the first frame was 100 and the transmission time of each frame instance was 9 time units. The release time of the first frame belonging to Class B was 80 with a transmission time of 11 time units. As mentioned

above, the observation window was 2 hyperperiods which was set to 2000. Therefore, during the observation time, there were 20 Class A frame instances and 10 frames belonging to Class B. Following the method described in Section 5.3.2, the parameters of the constant bandwidth server can be defined as: $C_{ser} = 17$ and $T_{ser} = 24$. The parameters for the compared credit-based shaper were: $iS_a = iS_b = 0.714$ and $sS_a = sS_b = -0.286$. The finish time of each event-triggered frame instance is depicted in Figure 5.9. The release time of the last Class A frame was 2000; therefore, there were only 19 frames that could be observed during the window. Results indicate that all event-triggered frames from different classes had similar finish times with both methods. The proposed method neither improved performance nor degraded it. When considering the transmission of delayed ST frames, although the conventional method also treats it successfully, the delayed frame treated by the proposed switch arrived at its destination with an earlier finish time with $D_R = -9.70$.

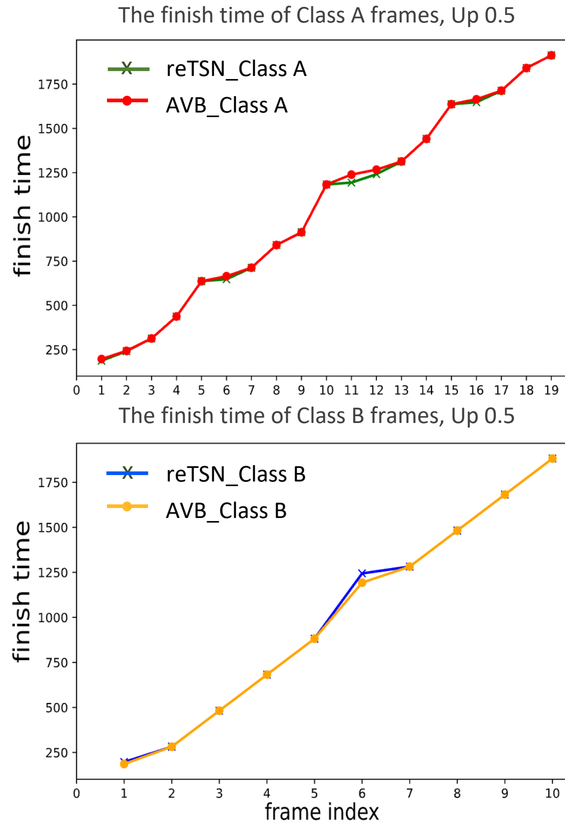

Figure 5.10: The finish time of event-triggered frames from Class A and Class B with $U_p = 0.5$ (index starts from 1)

Subsequently, we increased the network load and set the utilisation consumed by periodic traffic to approximately 0.5. The UUnifast-based generator randomly generated another specific case. We also considered 10 periodic flows with various periods [50, 200, 500, 500, 500, 50, 1000, 200, 200, 1000]. The hyperperiod was also 1000, and the transmission time of frame instances from relevant flows were [3, 10, 6, 52, 24, 3, 23, 3, 15, 12]. The precise utilisation of ST frames was 0.47. In this case, the randomly selected delayed ST frame was the $33rd$ frame instance in the GCL, with an actual release time 419 and a deadline 600, and it passed the acceptance test. The frame was also successfully transmitted by the AVB-based method. The first frame instance from Class A was released at 96, with 11 units transmission time, and the release time of the first Class B frame was 72, with a transmission time 10. The release time of the last Class A frame was 1996, and the finish time was definitely greater than 2000. Therefore, it could not be observed within the observation window. The parameters of the constant bandwidth server were $C_{cbs} = 17$ and $T_{cbs} = 32$. The parameters for the compared credit-based shaper were: $iS_a = iS_b = 0.53$ and $sS_a = sS_b = -0.47$. Figure 5.10 illustrates the finish time of each frame. Compared with the conventional AVB mechanism, the finish time of the five Class A frames was reduced slightly. Regarding the critical frames from Class B, although the performance of the two frames was slightly degraded, the finish time for the other frames was almost the same.

Figure 5.11 demonstrates the finish times of event-triggered frames when the utilisation consumed by periodic safety-critical traffic was set to 0.6. Their periods were [100, 1000, 200, 1000, 500, 200, 1000, 1000, 500, 100] with transmission time [3, 7, 20, 91, 5, 27, 24, 31, 12, 13]. The delayed ST frame was released at 740, and its deadline was 1000. It passed the acceptance test, and the proposed method can guarantee its transmission. However, the delayed frame could not have arrived at its destination within its deadline using the conventional method. During the observation time, the first Class A frame was ready to be transmitted at 55, and the first Class B frame arrived at 76. The transmission times of the frames from these two classes were 15 and 22, respectively. The parameters of constant bandwidth server were $C_{ser} = 23, T_{ser} = 58$. The parameters for the compared credit-based shaper were: $iS_a = iS_b = 0.412$ and $sS_a = sS_b = -0.588$. As Figure 5.11 shows, the finish times of most event-triggered frames from Class A were reduced. Although the portion of Class B frames that were degraded increased, the

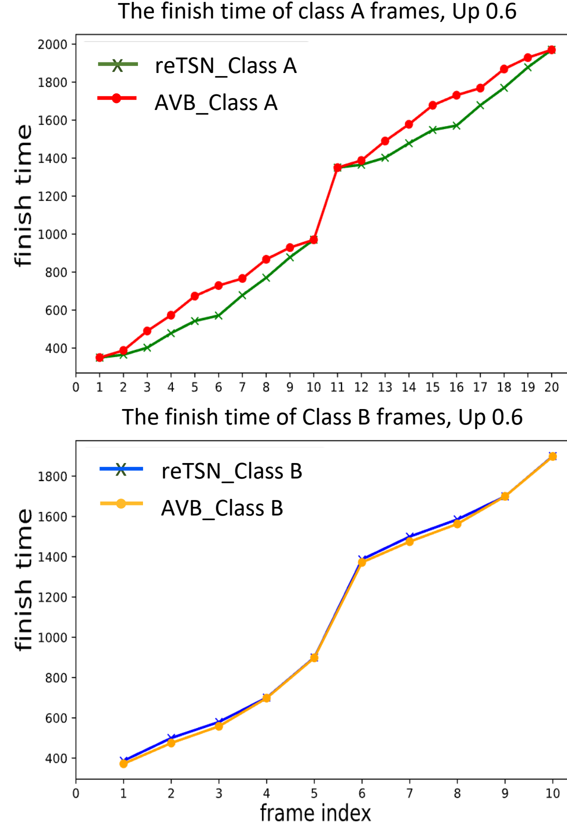

Figure 5.11: The finish time of event-triggered frames from Class A and Class B with $U_p = 0.6$ (index starts from 1)

increased time was not significant. Thus, the improvement of Class A frames was achieved with a minor impact on Class B frames.

In the highly loaded network with $U_p = 0.8$, the periods of 10 safety-critical flows were as follows: [100, 500, 200, 500, 500, 500, 500, 100, 500, 500] and the corresponding ST frames transmission time were [7, 45, 43, 18, 83, 23, 22, 6, 18, 9]. Neither method can guarantee the randomly selected delayed ST frame. The absolute release time of the first event-triggered frame in Class A was at 27, with a transmission time of 14. For the Class B frames, the first frame was released at 99 with 12 units of transmission time. The capacity of the constant bandwidth server C_{ser} was 24, and the period T_{ser} was 121. The parameters for the compared credit-based shaper were: $iS_a = iS_b = 0.219$ and $sS_a = sS_b = -0.781$. Only 7 event-triggered frames could be compared during the observation window. Frames are comparable if and only if a given frame's finish times in both methods are within the observation window. Figure 5.12

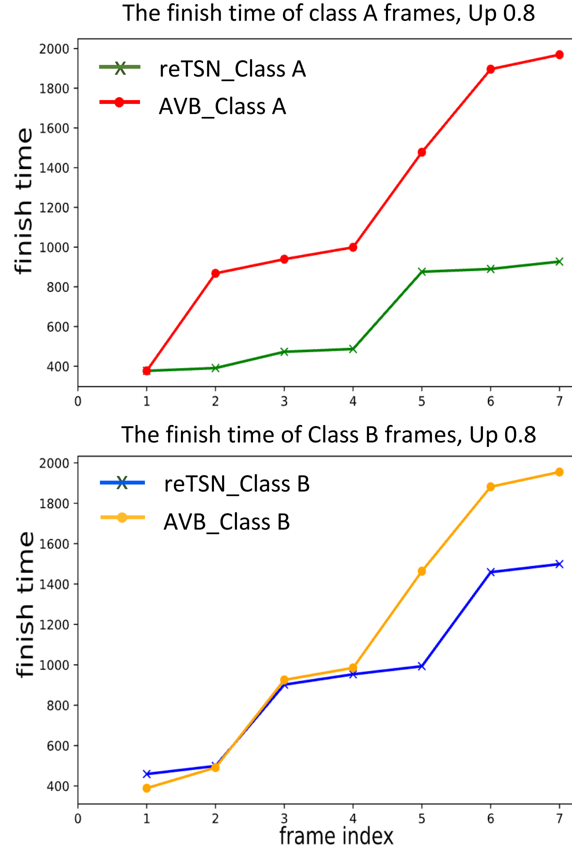

Figure 5.12: The finish time of event-triggered frames from Class A and Class B with $U_p = 0.8$ (index starts from 1)

shows that the constant server-based method dramatically reduced the finish time of the event-triggered frames in Class A and Class B; their performances improved simultaneously. The approach benefits from eliminating the postponed time caused by the credit replenishment procedure of the conventional method. Under a highly loaded network, the time to recharge the credit back to zero is increased by GCL-scheduled ST frames with a much higher probability. Although frames from Class A or B are already pending in the queue and the channel is idle, they cannot be transmitted because of their negative credits. However, in the proposed method, they could be dispatched immediately. Figure 5.6 illustrates the benefits of the proposed method intuitively.

In summary, under highly loaded network scenarios, the conventional credit-based-sharper-based switch cannot efficiently utilise the remaining bandwidth for event-triggered traffic transmission. In contrast, the proposed constant

bandwidth server-based switch could improve the utilisation of the remaining bandwidth and increase efficiency. At the same time, it could also improve the capability for temporal fault tolerance by reducing the likelihood of discarding delayed ST frames; thus, it enhances the safety of the system.

5.4.4 Larger-scale evaluation

In this experiment, using the same evaluation method as above, the simulation was used to run 10,000 trials under four scenarios. Overall, there were 300,000 critical frames and 10,000 delayed ST frames released within the observation window under each network load. The evaluation in the previous subsection demonstrated the differences in finish times for each observed frame. In the larger-scale evaluation, we focus on the overall performance improvements instead of analyzing each specific event-triggered traffic as done in the previous subsection. In each round of evaluation, the exact period of ST frames varies, while the parameter allocation rules for both the CBS and credit-based shapers remain consistent. Consequently, we will no longer illustrate the parameters for each round.

Table 5.3: The finish time variation of event-triggered frames with delayed ST frames

utilisation	Class	Improved (%)	Degraded (%)	Unchanged (%)	Mean reduced (%) (excluding zeros)	Median reduced (%) (excluding zeros)	Extra frames (by reTSN)
0.3	A	18.81	0.73	80.46	-4.77	-1.30	9
	B	6.91	19.94	73.15	+1.09	+0.89	0
0.5	A	43.08	1.66	55.26	-5.68	-2.74	795
	B	11.71	43.24	45.05	+3.61	+1.53	1
0.6	A	63.60	2.37	34.03	-9.73	-5.76	11063
	B	17.09	57.35	25.56	+4.77	+1.70	55
0.8	A	84.68	6.08	9.24	-23.49	-19.03	63518
	B	47.94	44.82	7.24	-2.15	-2.21	14764

Table 5.3 demonstrates the results of our large-scale experiment. With $U_p = 0.3$, the portions of unchanged frames from Class A and B were relatively high, at 80.46% and 73.15% respectively. The finish times of Class A frames were reduced by 4.77% on average. In contrast, for Class B frames, the finish time was increased by 1.09% on average, and the median value was 0.89%. In the network, with $U_p = 0.5$, the performance of 55.26% of the Class A frames and 45.05% Class B frames were unaffected. For the Class A frames,

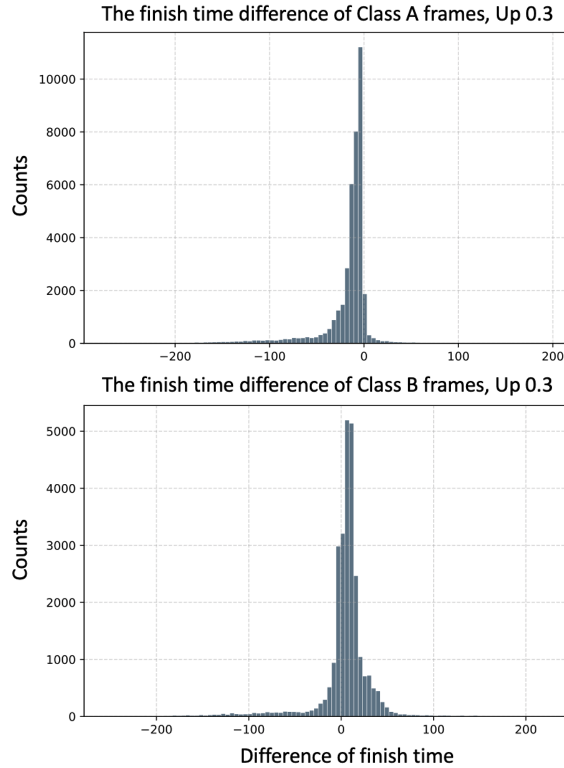

Figure 5.13: The finish time difference of event-triggered frames from Class A and Class B with network load $U_p = 0.3$

43.08% were improved with a smaller finish time, the values were reduced by 5.68% on average and the median reduced proportion was 2.74%. In contrast, 43.24% of the Class B frames had degraded performances with increased finish times, with an average increase of 3.61% and a median increase of 1.53%. Furthermore, within the observation window, 795 extra Class A frames finished their transmission, while the throughput of Class B traffic remains almost the same. In the medium-loaded network with $U_p = 0.6$, 63.60% of frames from Class A had a smaller finish time, and the average and median reduced proportions were 9.73% and 5.76%, respectively. However, only 17.09% of the frames from Class B were improved and the finish times were increased by 4.77% on average because 57.35% of the frames were degraded. This trend aligns with the results obtained from the specific case study. In lightly and moderately loaded networks the improvement of Class A performance was based on the sacrifice of Class B. However, it is worth noting that, 11,063 more Class A frames and 55 more Class B frames can finish their transmission

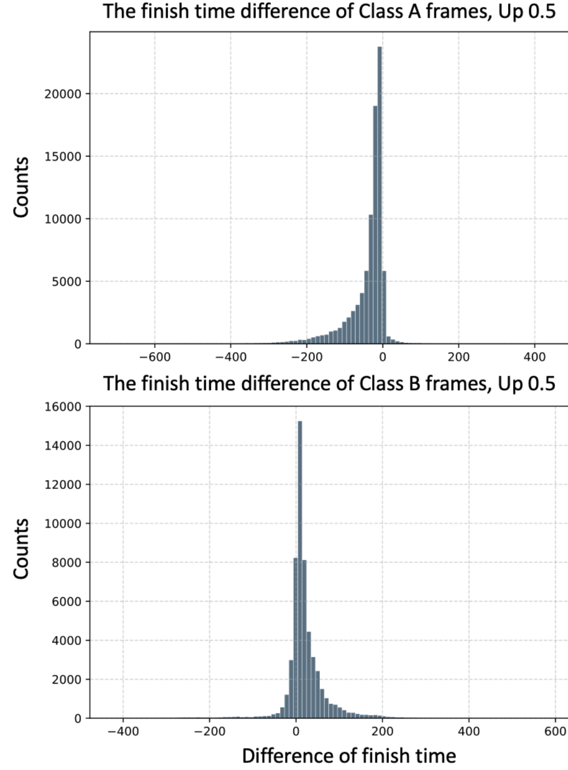

Figure 5.14: The finish time difference of event-triggered frames from Class A and Class B with network load $U_p = 0.5$

within the observation time when using the proposed method. The accepted delayed ST frames can postpone the transmission of event-triggered frames. In the highly loaded network, the proportions of the improved frames from both classes significantly increased to 84.68% and 47.94%, respectively. The finish time of comparable Class A frames decreased by 23.49% on average. For Class B, the finish times were also reduced by 2.15%. Further, 14,764 more Class B frames and 63,518 more Class A frames finished their transmission within the observation window. The throughput was increased by 31.76% and 14.76% for Class A and Class B traffic, respectively.

The histograms in Figure 5.13 illustrate the distribution of finish time differences of comparable event-triggered Class A and Class B frames in the network with $U_p = 0.3$. If the difference is negative, that means the performance improved and had an earlier finish time. Obviously, in Class A, the portion of improved frames is greater than the portion of degraded ones. In contrast, in Class B, the portion of degraded frames is larger than the portion

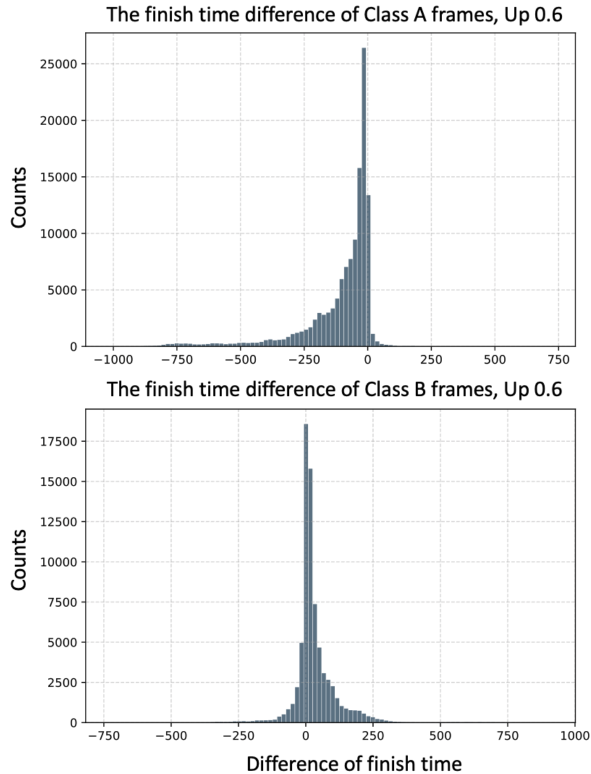

Figure 5.15: The finish time difference of event-triggered frames from Class A and Class B with network load $U_p = 0.6$

of improved ones.

Figure 5.14 and 5.15 illustrate the distribution of finish time differences when the utilisation of periodic frames was increased to $U_p = 0.5$ and $U_p = 0.6$. Compared with Figure 5.13, the distribution of both classes does not vary significantly. However, in the network with $U_p = 0.6$, the number of frames with earlier finish time from Class A increased significantly and matches the statistical results described in Table 5.3. Evidently, in the low and moderately loaded network, the promotion of the frames' performance (with shorter total transmission times) from Class A depends on the sacrifice of Class B. Additionally, delayed ST frames can interfere with the transmission of the frames from both classes. That is another reason why the finish time increased in some cases.

In the highly loaded network, the utilisation consumed by an ST periodic frame was 0.8. The benefit brought by the proposed method can be observed more clearly. Figure 5.16 illustrates the finish time difference distribution.

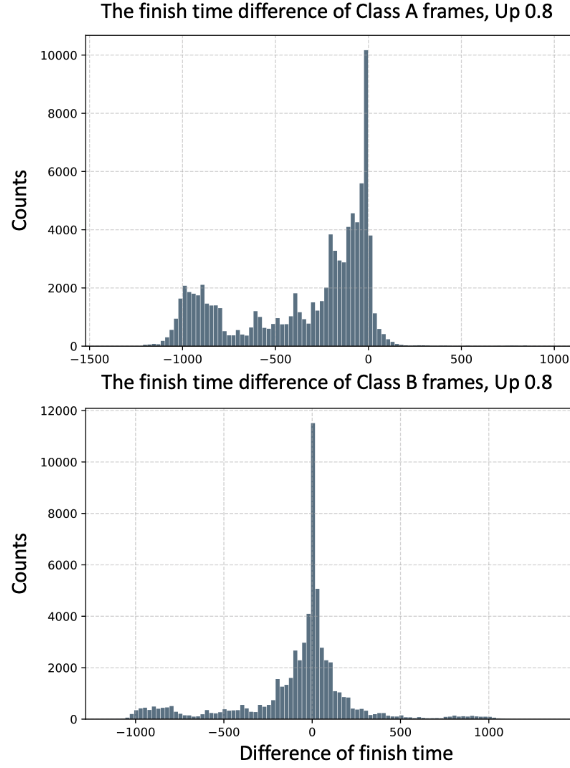

Figure 5.16: The finish time difference of event-triggered frames from Class A and Class B with network load $U_p = 0.8$

When compared with the histograms of low and moderately-loaded networks, the portion of improved Class B frames increased dramatically. The peak around -1000 can be explained by Figure 5.12 and Figure 5.6. Based on the AVB-based method, the last few frames were not allowed to be transmitted because of negative credit, and the credit increase was blocked by the transmission of ST frames.

According to the simulation results, the proposed method eliminates the time wasted by the credit replenishment procedure. The server-based deadline allocation method also prevents the channel from being occupied for a long-time by frames from the same class, and it improves the utilisation of the remaining bandwidth; it is more efficient, especially in a highly loaded network. At the same time, it also improves the capability of the ST frame's temporal fault tolerance by reducing the likelihood that delayed ST frames will be discarded.

5.5 Summary

In this chapter, we show that the proposed *Resilient and Efficient Time-Sensitive Networking (reTSN)* can efficiently utilise the bandwidth resources by eliminating the credit-recharging procedure. Furthermore, it can handle network uncertainties in two ways: (a) it can deal with more delayed safety-critical frames than the AVB switch to reduce drop-outs and associated potential safety-related problems. Compared to the conventional approach, our approach can schedule considerably more delayed ST frames even in a relatively highly loaded network; (b) For event-triggered traffic due to a dynamically changing environment and with unknown release times, reTSN can utilise the remaining bandwidth with higher efficiency, thus improving the performance of Class A frames with relatively higher priority. Especially in a highly loaded network, Class A frames' finish times (latency) were reduced by over 20% on average when compared to the AVB-based method. Furthermore, the throughput of event-triggered Class A frames was increased by 31.76%. In low and moderately loaded networks, the performance of Class B frames was degraded; however, the degradation was small and acceptable. Therefore, the proposed method can improve the resilience and efficiency of the network.

Chapter 6

Conclusions and Future Work

This final chapter's primary aim is to assess the work presented in the light of the primary research objectives, provide a summary of the thesis and highlight its contributions. This assessment is followed by a discussion of potential directions for future research.

The increasing number of computationally intensive functions, limitations of computational resources, dynamically changing driving scenarios, and heterogeneous hardware platforms are some key features of current autonomous systems. All of these factors can be barriers to the development and assurance of the system from both functional and non-functional perspectives, especially the uncertainties arising from them can threaten the safety of the system. This thesis contributes towards safe, time-critical, mixed-criticality adaptive autonomous systems through scheduling system resources, addressing the non-functional perspective, focusing on tackling timing faults and improving the system's resilience with highly efficient resource utilisation and awareness of multiple operational modes and system-wide functionality. To satisfy criticality-dependent timing requirements through scheduling resources, the research question can be summarised as: How to integrate functions with different criticalities on a constrained hardware platform whilst ensuring that all components comply with both functional and temporal requirements? And the non-functional safety challenges can be further summarised as follows:

- Execution/transmission of safety-critical tasks/data cannot be impacted by any non-safety-critical tasks/data;

- The execution/transmission of safety-critical tasks/data should be time predictable or deterministic;

- Timing faults of any safety-critical function should be tolerated;

- Shared resources for non-safety-critical tasks/data should be effectively managed to improve systems' quality of Service (QoS).

The hypothesis introduced earlier is based on understanding the challenges and is restated below:

"A suitable resources scheduling strategy can dramatically improve the resilience of safety-critical systems and achieve higher resource utilisation and system utility, even in the face of timing faults and operational mode changes."

The proposed timing-predictable and resource-efficient scheduling methods in this thesis can be understood from two main perspectives:

- **Timing-predictable:** The worst-case execution time estimates for each task are criticality-dependent, providing upper bounds that guarantee the system's safety at runtime. The proposed scheduling methods in this thesis ensure determinism for safety-critical tasks and data, offering predictability and certifiability to meet the timing requirements arising from non-functional safety requirements.

- **Resource-efficient:** The proposed methods in this thesis significantly improve the execution and transmission efficiency of non-safety-critical but mission-critical tasks and data, which is essential for maintaining the system's quality of service.

Overall, the proposed methods effectively schedule shared resources. Increasing the tolerance for timing faults in safety-critical tasks improves the system's resilience, while enhancing the performance of non-safety-critical tasks and data contributes to the system's survivability.

In this thesis, the research starts with designing task-level scheduling methods, which are then extended to the network level. This implies that our proposed methods can be applied to a wider range of shared resources, and the scenarios can be expanded beyond autonomous systems to other complex, large-scale environments. The next section summarises all of the contributions that provide evidence to support the above hypothesis.

6.1 Contributions

In this section, we re-emphasise the research contributions. Our proposed scheduling strategies are designed as a comprehensive solution to enhance time predictability and efficiency of hardware resource utilization, with a primary focus on autonomous systems. They are criticality-aware and aimed at addressing the timing faults arising from (1) task execution and (2) data transmission. These methods significantly enhance the resilience and resource efficiency of autonomous systems.

In Chapter 3, we proposed a context-aware graceful degradation strategy for mixed-criticality scheduling to improve the system's resilience in handling the uncertainties arising from conflicts for shared resources, based on causality analysis with awareness of multiple operational modes. Bayesian networks are adopted to introduce metrics from the functional domain into scheduling design in the non-functional domain. The proposed causality analysis-based degradation process "bridges the gap", so functional dependencies are considered in scheduling design and thus leads to a graceful degradation that is both feasible and reasonable in functional and non-functional terms. The system is able to continue to run with relatively high QoS during the degradation process in different operational modes while preserving more droppable tasks.

In Chapter 4, we describe a novel consistent static scheduling method that considers task precedence constraints and the survivability of low-criticality tasks. Unlike using different schedules to manage task execution in different system criticality modes and simply discarding droppable tasks to guarantee the execution of tasks with higher criticality levels, our proposed mixed-criticality DAG-based multi-core static scheduling method uses only one consistent schedule to safely manage the system in any mode when facing timing fault (i.e., overrun). This approach avoids the simultaneous discarding of a large number of droppable tasks, which would significantly reduce the system's QoS. Furthermore, having different schedules may potentially increase migration costs, as tasks may need to be executed on different cores after a mode change. Our approach eliminates migration costs and the need to check the safety of schedules during mode changes as there is no schedule change.

Both methods can guarantee the execution of safety-critical tasks, ensuring that tasks with higher criticality levels are executed without interference from lower criticality tasks when facing timing faults such as overruns. This improves the system's resilience. Additionally, the proposed mixed-criticality

scheduling methods take the practical execution behaviour into account, meaning that if a high-criticality task overruns its estimated execution time in normal mode $C(LO)$, it is likely that it will not execute up to its worst estimated execution in higher mode $C(HI)$ and the overrun may only be a small margin, which means that there may be sufficient slack time to support lower-criticality task execution. Both methods reduce the waste of slack time to preserve more droppable tasks, thereby significantly improving the system's survivability.

In Chapter 5, the scheduling problem is extended from computational to communication resources. This chapter highlights the challenges posed by limited bandwidth resources and the large amount of traffic with high bandwidth requirements in autonomous systems. To address these challenges, we adopt *Time-Sensitive Networks* (TSNs) as the foundation of our work, which allows the use of standard Ethernet as a real-time communication network and enables network sharing among multiple time-critical and non-time-critical applications. We propose a scheduling strategy (i.e., Resilient and efficient scheduling for Time-Sensitive Networking (reTSN)) that incorporates multi-level preemption, emergency queue techniques, acceptance tests, and constant bandwidth server-based run-time priority allocation to enhance the probability of temporal fault tolerance in the transmission of safety-critical traffic. This also improves bandwidth utilisation efficiency by reducing bandwidth waste for the transition of non-safety-critical traffic.

Summary

Overall, the scheduling problems of task execution and data transmission can be well addressed based on the strategies proposed in this thesis. An offline calculated fixed schedule (static task execution schedule for multi-core and gate-control list for scheduling network bandwidth) can provide determinism for safety-critical tasks/data and achieve temporal isolation to avoid interference from tasks/data with lower-criticality levels. Additionally, the graceful degradation strategy and server-based run-time priority allocation can efficiently use slack time to improve the survivability and QoS of non-safety critical tasks/data while achieving the primary goal of improving the system's resilience by tolerating timing faults. Thus the methods presented in this thesis meet the hypothesis, as evidenced in chapters 3-5. We have also demonstrated that these novel methods outperform established scheduling methods, in many situations, e.g. under heavy load, thus they contribute

to the state of the art of scheduling critical systems with shared resources.

6.2 Limitations and Constraints

Although the proposed methods meet the stated hypothesis, there are some limitations to each contribution. These are discussed in the rest of this section, leading to a discussion of future research.

The approach to Context-aware Graceful Degradation for Mixed-Criticality Scheduling uses fairly simple models of task criticality, etc. Considering more system factors would improve the definition of expected utility, e.g. modelling the interference from the dependency of task partitioning and allocation to the hardware platforms, to make the degradation order optimal from both functional and non-functional perspectives. Besides, this work is based on flat-structured scheduling methods, which means that a central scheduler is used to schedule all tasks in the entire system according to their fixed priority. The offline sensitivity analysis-based degradation is based on a too-pessimistic assumption that all HI criticality tasks overrun the same proportion at each test iteration because, based on a central scheduler, it is difficult to identify the tasks that can be affected by the overrun of a specific HI task. This leads to unnecessarily discarding some droppable tasks. Thus, although our approach is more effective than previously published methods, it could be refined further to improve system utilisation and QoS. To apply our methods in practice, constructing high-level Bayesian Belief Network (BBN) conditional probability tables in the automotive and aerospace industries can pose numerous challenges. This is attributed to the intricate and ever-changing characteristics of these domains, necessitating a meticulous selection of relevant conditions and constraints on complexity for effective data collection and training, ensuring sustainable development. For instance, challenges may arise in managing scenarios with a combination of adverse conditions and in determining the appropriate granularity of each variable's state. Although biases originating from a functional perspective might not directly result in timing faults, they can still lead to unmet expectations in terms of the expected Quality of Service (QoS).

For Resilience-aware Consistent Mixed-Criticality DAG Scheduling on Multicores, the time spent on GPU access and task execution is assumed to be included in the overall estimated execution time of tasks deployed on the

CPU cores. The dependencies between tasks with GPU requirements are represented by their access order and integrated into the system task graph. This simplifies the hardware model for scheduling design to a homogeneous multi-core architecture, enabling the scheduling of multi-periodic DAGs of mixed-criticality tasks on a multi-core platform. However, in this approach, task precedence constraints are only considered for offline schedule calculation, and system-wide functionality impact is not taken into account when facing timing faults. However, functional task-dependent factors and migration technology could be used to provide preference for dropping tasks with relatively lower system-wide functionality impact, thus maintaining higher functional QoS for the system. To apply our methods in practical scenarios, tasks must be amenable to modelling using Directed Acyclic Graphs (DAGs), and accurate estimation of Worst-Case Execution Time (WCET) across different criticality system modes is crucial. However, when dealing with advanced hardware platforms, particularly multi-core heterogeneous computational platforms, employing traditional static WCET analysis on real industrial programs poses numerous challenges for several reasons: (1) the intricate nature of real industrial programs, with a multitude of potential execution paths, presents a formidable challenge; (2) developing precise hardware models for novel architectures requires significant effort and a comprehensive hardware representation, which might not always be readily available. (3) the potential interference of shared hardware resources among concurrent tasks operating on multi-core heterogeneous architectures substantially escalates the complexity of timing analysis. Addressing these issues remains an ongoing research challenge.

reTSN is based on a simple centralised network structure, and verifying its effectiveness on a more complex network topology level would be necessary to fully validate the approach. When introducing more traffic categories, challenges may arise from the segmentation of server capacity based on the importance of different traffic types. Additionally, the efficiency of the proposed method in handling ST frames that arrive early needs to be validated. In future work, the implementation of reTSN on physical platforms (such as a mobile robot) could help to demonstrate its capability on real-world sensory data, including cameras and LiDARs. This would enable us to validate the simulation results and demonstrate the usefulness of reTSN in real-world applications. To implement our methods in practical scenarios, hardware support is required. For instance, a TSN-based switch capable of accommodating

multi-level preemption is essential. While the feasibility of this concept has been demonstrated, locating a suitable Commercial Off-The-Shelf (COTS) product remains challenging. Additionally, the current simulation tools available for TSN networks lack support for multi-level preemption. This implies that, given the significance of system safety, particularly in safety-critical contexts, prior to validating the proposed methods on actual hardware platforms, it might be necessary to develop a more comprehensive simulator or enhance existing tools for thorough validation. This process would involve covering as many scenarios as possible to ensure accurate evaluation.

6.3 Future Work

Despite the results presented here, there is still considerable potential to improve the presented scheduling strategies further. To be more specific, future work is discussed in the following subsections. Some of this work has already been studied but not thoroughly investigated due to time limitations.

6.3.1 Memory-aware Scheduling and Allocation

In this PhD project, the design of scheduling strategies is limited to the task-level, thus neglecting specific potential influencing factors. For instance, factors like memory, which can significantly impact the execution time of a task, are not taken into account explicitly. We believe that incorporating memory-related components could further enhance the performance of our proposed methods, and make it more practical.

In many automotive systems, the code for different applications is stored in inexpensive flash memory. Before a particular application is executed, its code is fetched from the flash to the on-chip memory located on the processor. The smaller the on-chip memory is, the more cost-effective the ECU is [39]. If the processor or memory access is not fast enough, the execution time might be too long to meet the desired sampling periods and sensor-to-actuator delays. In future work, minimising multi-level memory accesses, including on-chip and off-chip memory accesses, will be critical in reducing the execution time.

In recent years, integration trends, increased complexity in the memory system and the more sophisticated correlation between the execution of instructions and memory access have exacerbated the challenge of ensuring timing predictability for autonomous driving systems. Furthermore, integrating

heterogeneous software applications makes the issue more complicated. For instance, typical multi-core architectures consist of homogeneous or heterogeneous cores with multiple levels of coherent data and instruction caches connected using Network-on-Chip (NoC) for fast communication [119]. Scratchpad Memory (SPM) has been used as an alternative to caches in embedded systems due to energy efficiency, timing predictability, and scalability [15].

An SPM consists of an array of Static Random Access Memory (SRAM) cells. The SPM utilises a part of the memory address space. If the address falls within this specific address space, the corresponding data can be directly indexed into the SPM to access. They do not need tag arrays and comparators, which are essential to caches. Therefore, SPM is power-efficient. Multiple SPMs can maintain coherency among each other at the software level. Thus, for hardware area/power, the requirement for cache coherence can be eliminated. The disadvantage is that either the compiler or the programmer has to pay explicit attention to data allocation to the SPM to ensure efficiency. The programmer should know the latency for each memory access. Though SPM-based architectures can be used for timing predictability [119], data management will be the most challenging aspect of systems equipped with SPMs.

For future research on memory-aware scheduling and allocation, a formal definition of the data allocation problem for autonomous driving applications on SPM- and cache-based computational platforms should be proposed. Furthermore, the optimal scheduling and allocation strategies, which can improve application performance, reduce the WCET to some extent, and guarantee the time predictability of systems, will be critical contributions.

6.3.2 Real-world Case Study

In this thesis, the proposed scheduling methods have been evaluated using synthetic tasks. A real-world case study would provide additional support for the usability of the framework. Simulation can be more thorough and challenging than using a single physical system – but using a real-world case study helps both validate the methods and the simulation-based approach to evaluation. As introduced in Chapter 3, we have already set up a mobile robot using the Robot Operating System (ROS), which is used to integrate perception and control algorithms with a publisher-subscriber model. With the awareness of the importance of low latency in robot control, ROS 2 now supports real-time systems. The new Executors concept in ROS 2 can help

the system manage the execution flows of tasks [37]. The proposed mixed-schedule-based consistent schedule generation method can be used on top of ROS 2 executor to generate a static schedule for a mobile robot and guarantee the safety of the robot from a timing perspective. Such a system would provide a good basis for validating the methods developed in this thesis as it would be rich enough to reflect the challenges of autonomous driving systems whilst being simple enough to be feasible as a test and evaluation platform.

References

[1] AUTOSAR specification of operating system. v4.10.AUTOSAR. `http://www.autosar.org/,AUTOSAR,2010.`

[2] ISO 26262 road vehicles – functional safety. Nov 2011. International Organization for Standardization, Geneva, CH, Standard,.

[3] RTCA/DO-178B - Software Considerations in Airborne Systems and Equipment Certification. 1993. U.S. Dept. of Transportation, Federal Aviation Administration.

[4] T. Abdelzaher, S. Baruah, I. Bate, A. Burns, R. I. Davis, and Y. Hu. Scheduling classifiers for real-time hazard perception considering functional uncertainty. In *Proceedings of the 31st International Conference on Real-Time Networks and Systems*, pages 143–154, 2023.

[5] U. Abelein, H. Lochner, D. Hahn, and S. Straube. Complexity, quality and robustness-the challenges of tomorrow's automotive electronics. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 870–871. IEEE, 2012.

[6] K. Agrawal, S. Baruah, and A. Burns. Semi-clairvoyance in mixed-criticality scheduling. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 458–468. IEEE, 2019.

[7] I. Álvarez, I. Furió, J. Proenza, and M. Barranco. Design and experimental evaluation of the proactive transmission of replicated frames mechanism over time-sensitive networking. *Sensors*, 21(3):756, 2021.

[8] J. H. Anderson, S. Baruah, and B. B. Brandenburg. Multicore operating-system support for mixed criticality. In *Proceedings of the Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, volume 4, page 7. Citeseer, 2009.

[9] A. Ankan and A. Panda. pgmpy: Probabilistic graphical models using python. In *Proceedings of the 14th Python in Science Conference (SCIPY 2015)*. Citeseer, 2015.

[10] M. Ashjaei, M. Sjödin, and S. Mubeen. A novel frame preemption model in TSN networks. *Journal of Systems Architecture*, page 102037, 2021.

[11] M. K. Atiq, R. Muzaffar, Ó. Seijo, I. Val, and H.-P. Bernhard. When ieee 802.11 and 5g meet time-sensitive networking. *IEEE Open Journal of the Industrial Electronics Society*, 3:14–36, 2021.

[12] N. C. Audsley. *Optimal priority assignment and feasibility of static priority tasks with arbitrary start times.* Citeseer, 1991.

[13] N. C. Audsley. On priority assignment in fixed priority scheduling. *Information Processing Letters*, 79(1):39–44, 2001.

[14] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline-monotonic approach. *IFAC Proceedings Volumes*, 24(2):127–132, 1991.

[15] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No. 02TH8627)*, pages 73–78. IEEE, 2002.

[16] D. Barber. *Bayesian reasoning and machine learning.* Cambridge University Press, 2012.

[17] S. Baruah. The federated scheduling of systems of mixed-criticality sporadic dag tasks. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 227–236. IEEE, 2016.

[18] S. Baruah, V. Bonifaci, G. DAngelo, H. Li, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 145–154. IEEE, 2012.

[19] S. Baruah and A. Burns. Implementing mixed criticality systems in ada. In *Reliable Software Technologies-Ada-Europe 2011: 16th Ada-Europe*

International Conference on Reliable Software Technologies, Edinburgh, UK, June 20-24, 2011. Proceedings 16, pages 174–188. Springer, 2011.

[20] S. Baruah, B. Chattopadhyay, H. Li, and I. Shin. Mixed-criticality scheduling on multiprocessors. *Real-Time Systems*, 50(1):142–177, 2014.

[21] S. Baruah and G. Fohler. Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *2011 IEEE 32nd Real-Time Systems Symposium*, pages 3–12. IEEE, 2011.

[22] S. K. Baruah. Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors. *IEEE Transactions on Computers*, 53(6):781–784, 2004.

[23] S. K. Baruah, V. Bonifaci, G. d'Angelo, A. Marchetti-Spaccamela, S. v. d. Ster, and L. Stougie. Mixed-criticality scheduling of sporadic task systems. In *European symposium on algorithms*, pages 555–566. Springer, 2011.

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

[25] L. L. Bello, M. Ashjaei, G. Patti, and M. Behnam. Schedulability analysis of time-sensitive networks with scheduled traffic and preemption support. *Journal of Parallel and Distributed Computing*, 144:153–171, 2020.

[26] K. Bletsas, M. A. Awan, P. F. Souto, B. Akesson, A. Burns, and E. Tovar. Decoupling criticality and importance in mixed-criticality scheduling. In *Workshop on Mixed Criticality*, pages 25–32. York, 2018.

[27] M. Bouain, K. M. Ali, D. Berdjag, N. Fakhfakh, and R. B. Atitallah. An embedded multi-sensor data fusion design for vehicle perception tasks. *J. Commun.*, 13(1):8–14, 2018.

[28] J. Boudjadar, S. Ramanathan, A. Easwaran, and U. Nyman. Combining task-level and system-level scheduling modes for mixed criticality systems. In *2019 IEEE/ACM 23rd International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 1–10. IEEE, 2019.

[29] A. Burns and S. Baruah. Timing faults and mixed criticality systems. In *Dependable and Historic Computing*, pages 147–166. Springer, 2011.

[30] A. Burns and R. I. Davis. Schedulability analysis for adaptive mixed criticality systems with arbitrary deadlines and semi-clairvoyance. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 12–24. IEEE, 2020.

[31] A. Burns and R. I. Davis. Mixed criticality systems-a review:(february 2022). 2022.

[32] A. Burns, R. I. Davis, S. Baruah, and I. Bate. Robust mixed-criticality systems. *IEEE Transactions on Computers*, 67(10):1478–1491, 2018.

[33] A. Burns and A. J. Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001.

[34] S. Burton and J. A. McDermid. Closing the gaps: Complexity and uncertainty in the safety assurance and regulation of automated driving, 2023.

[35] G. C. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011.

[36] M. Çakır, T. Häckel, S. Reider, P. Meyer, F. Korf, and T. C. Schmidt. A QoS aware approach to service-oriented communication in future automotive networks. In *2019 IEEE Vehicular Networking Conference (VNC)*, pages 1–8. IEEE, 2019.

[37] D. Casini, T. Blaß, I. Lütkebohle, and B. Brandenburg. Response-time analysis of ros 2 processing chains under reservation-based scheduling. In *31st Euromicro Conference on Real-Time Systems*, pages 1–23. Schloss Dagstuhl, 2019.

[38] L. Chai, Q. Gao, and D. K. Panda. Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system. In *Seventh IEEE international symposium on cluster computing and the grid (CCGrid'07)*, pages 471–478. IEEE, 2007.

[39] S. Chakraborty, M. A. Al Faruque, W. Chang, D. Goswami, M. Wolf, and Q. Zhu. Automotive cyber–physical systems: A tutorial introduction. *IEEE Design & Test*, 33(4):92–108, 2016.

[40] G. Chen, N. Guan, D. Liu, Q. He, K. Huang, T. Stefanov, and W. Yi. Utilization-based scheduling of flexible mixed-criticality real-time tasks. *IEEE Transactions on Computers*, 67(4):543–558, 2017.

[41] W. M. D. Chia, S. L. Keoh, C. Goh, and C. Johnson. Risk assessment methodologies for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 2022.

[42] Y. Chu and A. Burns. Flexible hard real-time scheduling for deliberative AI systems. *Real-Time Systems*, 40(3):241–263, 2008.

[43] X. Dai. dag-gen-rnd: A randomized multi-DAG task generator for scheduling and allocation research, Mar. 2022.

[44] X. Dai, S. Zhao, Y. Jiang, X. Jiao, X. S. Hu, and W. Chang. Fixed-priority scheduling and controller co-design for time-sensitive networks. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–9, 2020.

[45] R. I. Davis and A. Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Systems*, 47(1):1–40, 2011.

[46] R. I. Davis and L. Cucu-Grosjean. A survey of probabilistic timing analysis techniques for real-time systems. *LITES: Leibniz Transactions on Embedded Systems*, pages 1–60, 2019.

[47] R. I. Davis, L. Cucu-Grosjean, M. Bertogna, and A. Burns. A review of priority assignment in real-time systems. *Journal of systems architecture*, 65:64–82, 2016.

[48] R. Dobrin, N. Desai, and S. Punnekkat. On fault-tolerant scheduling of time sensitive networks. In *4th International Workshop on Security and Dependability of Critical Embedded Real-Time Systems (CERTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[49] A. C. T. dos Santos, B. Schneider, and V. Nigam. TSNSCHED: Automated schedule generation for time sensitive networking. In *2019 Formal Methods in Computer Aided Design (FMCAD)*, pages 69–77. IEEE, 2019.

[50] S. Edgar and A. Burns. Statistical analysis of wcet for scheduling. In *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001)(Cat. No. 01PR1420)*, pages 215–224. IEEE, 2001.

[51] P. Ekberg and W. Yi. Schedulability analysis of a graph-based task model for mixed-criticality systems. *Real-time systems*, 52(1):1–37, 2016.

[52] P. Emberson and I. Bate. Extending a task allocation algorithm for graceful degradation of real-time distributed embedded systems. In *2008 Real-Time Systems Symposium*, pages 270–279. IEEE, 2008.

[53] P. Emberson, R. Stafford, and R. I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 6–11, 2010.

[54] A. Ermedahl. *A modular tool architecture for worst-case execution time analysis*. PhD thesis, Acta Universitatis Upsaliensis, 2003.

[55] A. Esper, G. Nelissen, V. Nélis, and E. Tovar. How realistic is the mixed-criticality real-time system model? In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, pages 139–148, 2015.

[56] J. Falk, D. Hellmanns, B. Carabelli, N. Nayak, F. Dürr, S. Kehrer, and K. Rothermel. NeSTiNg: Simulating IEEE time-sensitive networking (TSN) in OMNeT++. In *2019 International Conference on Networked Systems (NetSys)*, pages 1–8. IEEE, 2019.

[57] N. Finn. Introduction to time-sensitive networking. *IEEE Communications Standards Magazine*, 2(2):22–28, 2018.

[58] T. Fleming and A. Burns. Incorporating the notion of importance into mixed criticality systems. In *Proc. 2nd Workshop on Mixed Criticality Systems (WMC), RTSS*, pages 33–38, 2014.

[59] A. Frigerio, B. Vermeulen, and K. Goossens. Automotive architecture topologies: Analysis for safety-critical autonomous vehicle applications. *IEEE Access*, 2021.

[60] N. Fumio, Y. Asada, T. Sobue, M. Yano, O. Sakanoue, K. Maeda, and M. Saito. Vehicle electronic control units for autonomous driving in safety and comfort. *Hitachi Review*, 71(1), 2022.

[61] M. Gadd, D. De Martini, L. Marchegiani, P. Newman, and L. Kunze. Sense–assess–explain (SAX): Building trust in autonomous vehicles in challenging real-world driving scenarios. In *2020 IEEE Intelligent Vehicles Symposium (IV)*, pages 150–155. IEEE, 2020.

[62] M. Glinz. On non-functional requirements. In *15th IEEE international requirements engineering conference (RE 2007)*, pages 21–26. IEEE, 2007.

[63] P. Graydon and I. Bate. Safety assurance driven problem formulation for mixed-criticality scheduling. *Proc. WMC, RTSS*, pages 19–24, 2013.

[64] P. Graydon and I. Bate. Realistic safety cases for the timing of systems. *The Computer Journal*, 57(5):759–774, 2014.

[65] X. Gu and A. Easwaran. Dynamic budget management with service guarantees for mixed-criticality systems. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 47–56. IEEE, 2016.

[66] N. Guan and Z. Dong. Industry challenge.

[67] F. He, L. Zhao, and E. Li. Impact analysis of flow shaping in ethernet-avb/tsn and afdx from network calculus and simulation perspective. *Sensors*, 17(5):1181, 2017.

[68] W. Hess, D. Kohler, H. Rapp, and D. Andor. Real-time loop closure in 2d lidar slam. In *2016 IEEE international conference on robotics and automation (ICRA)*, pages 1271–1278. IEEE, 2016.

[69] R. Hofmann, B. Nikolić, and R. Ernst. Challenges and limitations of IEEE 802.1 CB-2017. *IEEE Embedded Systems Letters*, 12(4):105–108, 2019.

[70] S. Holzknecht, E. Biebl, and H.-U. Michel. Graceful degradation for driver assistance systems. In *Advanced Microsystems for Automotive Applications 2009*, pages 255–265. Springer, 2009.

[71] Y. Hotta, A. Inoue, H. Bessho, C. Mangin, and R. Kawate. Experimental study of a low-delay Ethernet switch for real time networks. In *16th IEEE Int. Conf. on High Performance Switching and Routing*, 2015.

[72] T. Ishigooka, S. Otsuka, K. Serizawa, R. Tsuchiya, and F. Narisawa. Graceful degradation design process for autonomous driving system. In *International Conference on Computer Safety, Reliability, and Security*, pages 19–34. Springer, 2019.

[73] P. Jiang, D. Ergu, F. Liu, Y. Cai, and B. Ma. A review of yolo algorithm developments. *Procedia Computer Science*, 199:1066–1073, 2022.

[74] Z. Jiang, S. Zhao, R. Wei, D. Yang, R. Paterson, N. Guan, Y. Zhuang, and N. Audsly. Bridging the pragmatic gaps for mixed-criticality systems in the automotive industry. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[75] G. Jocher. YOLOv5. accessed 2022-10-04.

[76] A. V. Kanhere and G. X. Gao. Lidar slam utilizing normal distribution transform and measurement consensus.

[77] O. R. Kelly, H. Aydin, and B. Zhao. On partitioned scheduling of fixed-priority mixed-criticality task sets. In *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1051–1059. IEEE, 2011.

[78] U. Khan and I. Bate. Wcet analysis of modern processors using multi-criteria optimisation. In *2009 1st International Symposium on Search Based Software Engineering*, pages 103–112. IEEE, 2009.

[79] Y. Kim. Very low latency packet delivery requirements and problem statements. In *IEEE 802.1 AVB Task Group Interim Meeting. Atlanta, GA USA, Nov 2011*, 2011.

[80] U. B. Kjærulff and A. L. Madsen. Probabilistic networks-an introduction to bayesian networks and influence diagrams. *Aalborg University*, pages 10–31, 2005.

[81] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

[82] J. Lee, H. S. Chwa, L. T. Phan, I. Shin, and I. Lee. Mc-adapt: Adaptive task dropping in mixed-criticality scheduling. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):1–21, 2017.

[83] J. Lee and S. Park. Time-sensitive network (tsn) experiment in sensor-based integrated environment for autonomous driving. *Sensors*, 19(5):1111, 2019.

[84] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance evaluation*, 2(4):237–250, 1982.

[85] J. Li, K. Agrawal, C. Lu, and C. Gill. Outstanding paper award: Analysis of global edf for parallel tasks. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 3–13. IEEE, 2013.

[86] J. Li, D. Ferry, S. Ahuja, K. Agrawal, C. Gill, and C. Lu. Mixed-criticality federated scheduling for parallel real-time tasks. *Real-time systems*, 53(5):760–811, 2017.

[87] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[88] D. Liu, N. Guan, J. Spasic, G. Chen, S. Liu, T. Stefanov, and W. Yi. Scheduling analysis of imprecise mixed-criticality real-time tasks. *IEEE Transactions on Computers*, 67(7):975–991, 2018.

[89] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022.

[90] J. A. McDermid, Y. Jia, and I. Habli. Towards a framework for safety assurance of autonomous systems. In *Artificial Intelligence Safety 2019*, pages 1–7. CEUR Workshop Proceedings, 2019.

[91] R. Medina, E. Borde, and L. Pautet. Directed acyclic graph scheduling for mixed-criticality systems. In *Ada-Europe International Conference on Reliable Software Technologies*, pages 217–232. Springer, 2017.

[92] R. Medina, E. Borde, and L. Pautet. Scheduling multi-periodic mixed-criticality dags on multi-core architectures. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 254–264. IEEE, 2018.

[93] R. Medina, E. Borde, and L. Pautet. Generalized mixed-criticality static scheduling for periodic directed acyclic graphs on multi-core processors. *IEEE Transactions on Computers*, 70(3):457–470, 2020.

[94] M. Mody. Adas front camera: Demystifying resolution and frame-rate. *EE News*, 2016.

[95] M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos. Mixed-criticality real-time scheduling for multicore systems. In *2010 10th IEEE international conference on computer and information technology*, pages 1864–1871. IEEE, 2010.

[96] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardos. Orb-slam: a versatile and accurate monocular slam system. *IEEE transactions on robotics*, 31(5):1147–1163, 2015.

[97] A. Nasrallah, A. S. Thyagaturu, Z. Alharbi, C. Wang, X. Shao, M. Reisslein, and H. Elbakoury. Performance comparison of ieee 802.1 tsn time aware shaper (tas) and asynchronous traffic shaper (ats). *IEEE Access*, 7:44165–44181, 2019.

[98] A. Nguyen and B. Le. 3d point cloud segmentation: A survey. In *2013 6th IEEE conference on robotics, automation and mechatronics (RAM)*, pages 225–230. IEEE, 2013.

[99] M. A. Ojewale, P. M. Yomsi, and B. Nikolić. Multi-level preemption in TSN: Feasibility and requirements analysis. In *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*, pages 47–55. IEEE, 2020.

[100] R. S. Oliver, S. S. Craciunas, and W. Steiner. IEEE 802.1 Qbv gate control list synthesis using array theory encoding. In *2018 IEEE Real-*

Time and Embedded Technology and Applications Symposium (RTAS), pages 13–24. IEEE, 2018.

[101] M. Osborne, R. D. Hawkins, and J. A. McDermid. Analysing the safety of decision-making in autonomous systems. In *SAFECOMP 2022 (41st International Conference on Computer Safety, Reliability and Security)*. York, 2022.

[102] B. Paden, M. Čáp, S. Z. Yong, D. Yershov, and E. Frazzoli. A survey of motion planning and control techniques for self-driving urban vehicles. *IEEE Transactions on intelligent vehicles*, 1(1):33–55, 2016.

[103] R. M. Pathan. Improving the schedulability and quality of service for federated scheduling of parallel mixed-criticality tasks on multiprocessors. In *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[104] S. D. Pendleton, H. Andersen, X. Du, X. Shen, M. Meghjani, Y. H. Eng, D. Rus, and M. H. Ang. Perception, planning, control, and coordination for autonomous vehicles. *Machines*, 5(1):6, 2017.

[105] J. Pisarov and G. Mester. The future of autonomous vehicles. *FME Transactions*, 49(1):29–35, 2021.

[106] K. Rehrl and S. Gröchenig. Evaluating localization accuracy of automated driving systems. *Sensors*, 21(17):5855, 2021.

[107] J. Reich, M. Wellstein, I. Sorokos, F. Oboril, and K.-U. Scholl. Towards a software component to perform situation-aware dynamic risk assessment for autonomous vehicles. In *European Dependable Computing Conference*, pages 3–11. Springer, 2021.

[108] J. Ren and L. T. X. Phan. Mixed-criticality scheduling on multiprocessors using task grouping. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 25–34. IEEE, 2015.

[109] L. Rierson. *Developing safety-critical software: a practical guide for aviation software and DO-178C compliance*. CRC Press, 2017.

[110] D. Rose. The four pillars of self-driving cars, 2018.

[111] S. Samii and H. Zinner. Level 5 by layer 2: Time-sensitive networking for autonomous vehicles. *IEEE Communications Standards Magazine*, 2(2):62–68, 2018.

[112] L. Sha, T. Abdelzaher, K.-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real time scheduling theory: A historical perspective. *Real-time systems*, 28:101–155, 2004.

[113] T. Steinbach, K. Müller, F. Korf, and R. Röllig. Real-time Ethernet in-car backbones: First insights into an automotive prototype. In *2014 IEEE Vehicular Networking Conference (VNC)*, pages 133–134. IEEE, 2014.

[114] W. Steiner, S. S. Craciunas, and R. S. Oliver. Traffic planning for time-sensitive communication. *IEEE Communications Standards Magazine*, 2(2):42–47, 2018.

[115] T. Stüber, L. Osswald, S. Lindner, and M. Menth. A survey of scheduling algorithms for the time-aware shaper in time-sensitive networking (tsn). *IEEE Access*, 2023.

[116] N. Suthar, P. Indr, and P. Vinit. A technical survey on dbscan clustering algorithm. *Int. J. Sci. Eng. Res*, 4:1775–1781, 2013.

[117] D. Thiele and R. Ernst. Formal worst-case performance analysis of time-sensitive Ethernet with frame preemption. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–9. IEEE, 2016.

[118] P. Trucco, E. Cagno, F. Ruggeri, and O. Grande. A bayesian belief network modelling of organisational factors in risk analysis: A case study in maritime transportation. *Reliability Engineering & System Safety*, 93(6):845–856, 2008.

[119] V. Venkataramani, M. C. Chan, and T. Mitra. Scratchpad-memory management for multi-threaded applications on many-core architectures. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(1):1–28, 2019.

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

[121] J. Wang, J. Liu, and N. Kato. Networking and communications in autonomous driving: A survey. *IEEE Communications Surveys & Tutorials*, 21(2):1243–1274, 2018.

[122] J. Wegener, H. Sthamer, B. F. Jones, and D. E. Eyres. Testing real-time systems using genetic algorithms. *Software Quality Journal*, 6:127–135, 1997.

[123] S. Wheeler, I. Bate, and M. Bartlett. Video subset selection for measurement based worst case execution time analysis. In *2011 6th IEEE International Symposium on Industrial and Embedded Systems*, pages 213–222. IEEE, 2011.

[124] J. Wiedmann. Implementation and evaluation of trace-based timing analysis. 2019.

[125] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

[126] L. Zhao, P. Pop, Z. Zheng, H. Daigmorte, and M. Boyer. Latency analysis of multiple classes of AVB traffic in TSN with standard credit behavior using network calculus. *IEEE Transactions on Industrial Electronics*, 2020.

[127] S. Zhao, X. Dai, I. Bate, A. Burns, and W. Chang. DAG scheduling and analysis on multiprocessor systems: Exploitation of parallelism and dependency. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 128–140. IEEE, 2020.

[128] D. Ziegenbein, S. Saidi, X. S. Hu, and S. Steinhorst. Future Automotive HW/SW Platform Design (Dagstuhl Seminar 19502). *Dagstuhl Reports*, 9(12):28–66, 2020.