High Performance Real-Time Scheduling Framework for Multiprocessor Systems

Nan Chen

PhD

University of York

Computer Science

October 2023

Abstract

Embedded systems, performing specific functions in modern devices, have become pervasive in today's technology landscape. As many of these systems are real-time systems, they necessitate operations with stringent time constraints. This is especially evident in sectors like automotive and aerospace. This thesis introduces a High Performance Real-time Scheduling (HPRTS) framework, which is designed to navigate the multifaceted challenges faced by multiprocessor real-time systems.

To begin with, the research attempts to bridge the gap between system reliability and resource sharing in Mixed-Criticality Systems (MCS). In addressing this, a novel fault-tolerance solution is presented. Its main goal is to enhance fault management and reduce blocking time during fault tolerance. Following this, the thesis delves into task allocation in systems with shared resources. In this context, we introduce a distinct Resource Contention Model (RCM). Using this model as a foundation, our allocation strategy is formulated with the aim to reduce resource contention. Moreover, in light of the escalating system complexity where tasks are represented using Directed Acyclic Graph (DAG) models, the research unveils a new Response Time Analysis (RTA) for multi-DAG systems. This particular analysis has been tailored to provide a safe and more refined bound.

Reflecting on the contributions made, the achievements of the thesis highlight the potency of the HPRTS framework in steering real-time embedded systems toward high performance.

Contents

List of Tables

List of Figures

Acknowledgment

I extend my heartfelt gratitude to the University of York for seven transformative years, transitioning from undergraduate to doctoral studies.

My deepest appreciation goes to my supervisors: Prof. Alan Burns, Prof. Wanli Chang, and Dr. Siyuan Ji. Their unwavering support and expertise have been instrumental in my academic journey.

I am profoundly thankful to Dr. Zhishan Guo for his role as my external examiner. His insights and thorough examination have significantly contributed to the depth and quality of this research.

I am also grateful to the Real-Time Distributed Systems group, including Ian Gray, Iain Bate, Shuai Zhao, Xiaotian Dai, Jie Zou, Haitong Wang, and others. Their collaboration and feedback have been invaluable. Special mention to Shuai Zhao for our joint achievements and to Ian Gray for his insightful thesis review.

To my parents, your unwavering support and sacrifices have been the foundation of my academic success. I also thank my friends: Xinyi Hu, Zhiheng Hu, and Qianyi Xu, for their constant emotional support.

In closing, my sincere thanks to everyone who contributed to this research. Your collective efforts have made this thesis possible.

Declaration

I declare that the work presented in this thesis is entirely my own creation. While some portions of the content have been featured in published articles, submissions, and other formal documents, they are explicitly detailed as follows:

- W. Chang, S. Zhao, S. Burton, T. Chen, N. Chen, N. Audsley. Hardware/-Software Co-Synthesis and Co-Optimization for Autonomous Systems. *Design Automation Conference (DAC)*. 2021.

- N. Chen, S. Zhao, I. Gray, A. Burns, S. Ji, W. Chang. MSRP-FT: Reliable Resource Sharing on Multiprocessor Mixed-Criticality Systems. *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2022

- N. Chen, S. Zhao, I. Gray, A. Burns, S. Ji, W. Chang. Precise Response Time Analysis for Multiple DAG Tasks with Intra-task Priority Assignment. *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2023.

- W. Chang, N. Chen, S. Zhao, X. Dai. *Evolvement of Scheduling Theories for Autonomous Vehicles*. Book Chapter. Springer. 2023.

- G. Xie, Y. Zhang, N. Chen, W. Chang. A High-Flexibility CAN-TSN Gateway with a Low-Congestion TSN-to-CAN Scheduler. *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*. 2023.

- S. Zhao, N. Chen, Y. Fang, Z. Li, W. Chang. A Universal Method for Task Allocation on FP-FPS Multiprocessor Systems with Spin Locks. *Design Automation Conference (DAC)*. 2023.

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

Chapter 1

Introduction

Embedded systems are computer systems integrated into other devices or products to perform specific functions. According to [90], embedded systems can be described as a combination of hardware and software components that work together to control the device or product it is integrated with. The hardware component includes a microprocessor or microcontroller and various peripheral devices, such as sensors, actuators, and memory. The software component includes an operating system and applications, which control the hardware and provide the necessary functionality for the system to perform its designated task. As the field of embedded systems continues to evolve, new developments in technology are driving innovation in this area. For example, the increasing use of wireless communication technologies, such as Wi-Fi and Bluetooth, and the growth of the Internet of Things (IoT) are enabling the development of new types of embedded systems that can interact with other devices and systems [85].

Many embedded systems are designed to operate in real-time environments. Real-time systems are characterized by their ability to respond to inputs and events within strict time constraints, making them ideal for use in critical applications where timely and accurate processing is required [30]. The integration of embedded systems with real-time capabilities is critical in industries, such as automotive, aerospace, and telecommunications [46]. For example, in the automotive industry, embedded systems are used in real-time to control various systems within the vehicle, such as engine management and safety systems, to ensure reliable and efficient operation [32].

In a real-time system, tasks that fail to complete within the allotted time

are referred to as *deadline miss*. According to the cost of the outcome, real-time systems can be broadly categorized as *hard* or *soft*. Hard real-time systems have zero tolerance for *deadline miss* and the occurrence of one will result in the complete failure of the system and have catastrophic consequences. Conversely, *deadline miss* in soft real-time systems generally do not result in severe outcomes, but persistent missed deadlines will affect the Quality of Service (QoS) of the system. Taking the example of an embedded system in an automobile, a *deadline miss* in the multimedia system will only result in a subpar customer experience. However, if the Anti-lock Braking System (ABS), which prevents the vehicle from losing control due to locked brakes, experiences a *deadline miss*, the consequences can be devastating, potentially leading to loss of life.

1.1 Motivation

Driven by applications such as autonomous vehicles, spacecraft, robotics, and industrial automation, real-time systems are required to implement ever more complex functionalities with high performance while maintaining conventional timing predictability, reliability, and cost-effectiveness. These systems often contain a large number of shared resources, including elements like memory sections, data structures, Input/Output (I/O) devices, storage media, network sockets, and hardware components, all of which require coordinated mechanisms such as locks and semaphores [50] to ensure mutually exclusive access. This coordination is crucial for maintaining data integrity and system stability while preventing conflicts and ensuring efficient, reliable operations. However, the need for mutually exclusive access to shared resources can lead to blocking due to contention. In this context, the conventional requirements of timing predictability and reliability still need to be satisfied. That is, the deadlines of tasks must be met while failures during task executions must be resolved which is particularly hard.

Several multiprocessor resource-sharing protocols have been proposed to bound and minimize blocking time, including Multiprocessor Stack Resource Protocol (MSRP) [41] and Multiprocessor resource sharing Protocol (MrsP) [22]. However, reliability has not been accounted for i.e., handling task faults, which is imperative in mission-critical scenarios. Existing fault-tolerance methods are based on redundancy, and they may be directly applied to shared resources by scheduling repeated task executions and resource accesses a sufficient number of times to get the correct output. However, this leads to severe resource contention and undermines system schedulability.

To address the issue of resource contention across processors, there has been significant focus on resource-aware task allocation in fully-partitioned multiprocessor systems. There are many resource-aware task allocation methods available, as cited in [48,49,55,62,97]. These methods cater to various scheduling techniques and resource-sharing protocols, such as Fixed-Priority Scheduling (FPS) and the MSRP [42]. Unlike conventional allocation methods such as Worst-Fit (WF) and Best-Fit (BF), these methods focus on discovering the resource-access relationships between tasks and reduces contention by localizing the shared resources. They can be generally categorized into two major approaches, which execute critical sections (i) on the *host processor* of tasks or (ii) on *dedicated processors*. The first approach aims to allocate tasks with the same shared resources to one processor to reduce contention. However, for complex systems with intensive resource sharing, the methods in this approach become ineffective due to the utilization of coarse-grained analysis of contention levels between tasks [49,62,97]. The second approach circumvents the aforementioned issues by designating a set of processors specifically for resource access. Yet, this approach relies on a search-based method, aided by schedulability tests, which results in high computation times. Additionally, it introduces extra migration overhead

and is less advantageous with frequent resource accesses [109].

Furthermore, as system complexity increases, tasks may need to adhere to specific execution orders due to functional dependencies between them. These dependencies are represented using Directed Acyclic Graph (DAG) models. For optimal performance, multiple DAGs must be scheduled, and their Worst-Case Response Times (WCRT) must be bounded. This schedulability analysis during the design phase is crucial for estimating the necessary hardware resources. Research into the Response Time Analysis (RTA) of DAGs encompasses a broad spectrum of scheduling schemes, including global [38, 74], fully-partitioned [37], and federated [9]. When focusing on a typical setup where multiple DAGs operate on a homogeneous multi-core system using a global scheme, there have been significant research efforts aimed at establishing bounds for worst-case execution scenarios through RTA. Broadly, the existing analysis techniques fall into two categories: the *generic bound* [36, 74, 86] and the *priority-explicit bound* [53, 54, 107, 108]. Generic bounds can be overly conservative without knowledge of the intra-task (node-level) priority. In contrast, the priority-explicit bound, which is based on intra-task priority knowledge, reduces unnecessary interference and blocking calculations. However, it retains some pessimism due to the overlooked parallelism between nodes. Given the pessimistic nature of current DAG analysis methods, system engineers often resort to substantial resource over-provisioning. This tendency has become a significant hurdle for implementing the RTA of DAG tasks in real-world systems.

1.2 Thesis Aim

The aforementioned limitations hinder the advancement of real-time systems towards high performance. In this thesis, we introduce a High Performance Real-Time Scheduling (HPRTS) framework. This framework tackles intricate

challenges in multiprocessor architectures with FPS scheduling, including reliable resource sharing, contention-aware allocation, and RTA for multi-DAG systems—all crucial for propelling real-time systems to higher performance levels.

The initial step of the proposed HPRTS framework seeks to bridge the gap between reliability and resource sharing. We focus on Mixed-Criticality Systems (MCS), which are prevalent in practical applications [59, 101]. Their intricate execution model makes the problem more challenging to address. We introduce the first fault-tolerant solution for multiprocessor MCS with shared resources. This proposed solution should be suitable for MCS with any number of criticality levels. New fault-tolerant techniques must be proposed to efficiently handle faults in the presence of shared resources, with the aim of reducing blocking time during fault tolerance. Concurrently, a schedulability test must be introduced to ensure the timing predictability of the proposed solution.

In the second step of the HPRTS framework of the thesis, we address the task allocation problem in the presence of shared resources. To mitigate resource contention in multiprocessor systems, a comprehensive contention-aware model is essential. This model should accurately capture task resource-access behaviors, guiding allocation to minimize contention effectively. It must be efficient, avoiding the need for repeated schedulability tests, ensuring both reduced computational costs and adaptability for run-time applications. The allocation strategy will prevent tasks from migrating, thus avoiding additional migration overheads.

In the concluding step of the proposed HPRTS, we embark on an exploration of a sophisticated task model. The objective is to curtail resource over-provisioning during the design phase of systems employing DAG tasks, recognizing that such over-provisioning can hinder the pursuit of high performance in real-time systems. Our attention is centered on systems equipped with knowl-

edge at the node level, a feature that is directly applicable in practical scenarios. Following this, we present an analytical bound through RTA tailored to encapsulate the worst-case execution scenarios, thereby safeguarding the system's timing predictability. The proposed analytical bound is crafted to explore the parallelism inherent in the execution of DAG tasks. It aims to minimize unnecessary interference and blocking delay computations, leading to a reduction in pessimistic estimations and effectively addressing the challenges of resource over-provisioning.

1.3 Thesis Hypothesis

The thesis hypothesis posits that integrating reliable resource sharing, contention-aware task allocation, and RTA for multi-DAG systems will significantly enhance real-time embedded system performance. This hypothesis is grounded in the following assumptions:

Reliable resource sharing: While resource contention complicates the adherence to timing constraints in real-time embedded systems, current fault-tolerance methods further intensify this contention, thereby diminishing system schedulability. Our proposed fault-tolerant solution aims to shorten the fault-tolerance process, thereby improving system schedulability over existing methods by at least 10%.

Contention-aware task allocation: Effective task allocation is pivotal for systems with intensive resource sharing. Our proposed algorithm, cognizant of task resource usage, is designed to diminish resource contention and outperform existing allocation methods in enhancing system performance by alleviating the computation speed and schedulability by at least 10%.

RTA for multi-DAG systems: Applications in modern real-time embedded systems are often modeled as DAG tasks. A precise RTA bound can mitigate

hardware resource over-provisioning challenges. Our proposed RTA for multi-DAG systems is designed to offer a safe and tight bound, scheduling at least 10% more systems than current bounds.

In summary, the hypothesis suggests that by synergizing these three components, the overall efficiency and efficacy of real-time embedded systems will see marked improvement, leading to superior performance. Experimental and simulation methods will be employed to validate this hypothesis.

1.4 Success Criteria and Contributions

To evaluate the effectiveness of the proposed framework in this thesis, a set of Success Criterias (SC) have been established. To validate the hypothesis stated in Section 1.3, the following must be developed:

SC-1: A system execution model compatible with an arbitrary number of criticality levels is constructed in which faults occurring in normal sections (i.e., without shared resources accessed) and critical sections (i.e., with shared resources accessed) are treated separately.

SC-2: A novel protocol is proposed to address faults during accessing shared resources, aiming to minimize blocking time following a schedulability analysis to provide the timing guarantees.

SC-3: A Resource Contention Model (RCM) that approximates the degree of resource contention between tasks in the context of FIFO-spin locks.

SC-4: A contention-ware task allocation algorithm based on the RCM, where shared resources are accessed by tasks from their host processor.

SC-5: An RTA for multi-DAG systems, utilizing novel analysis techniques, effectively harnesses node-level parallelism. It safely eliminates redundant calculations and workloads that cannot contribute to delays, hence ensures a tighter bound.

1.5 Thesis Outline

The thesis is structured as follows:

Chapter 2: This chapter offers an in-depth overview of real-time systems, covering task classifications, scheduling methodologies, and the intricacies of multiprocessor environments. It places emphasis on synchronization in resource sharing, the role of MCS in scheduling, and the evolution of task allocation. The importance of DAG tasks is underscored, and current research limitations are addressed, paving the way for deeper exploration in the subsequent chapters.

Chapter 3: Here, the first fault-tolerant solution tailored for multiprocessor MCS with shared resources is introduced. A system execution model compatible with any number of criticality levels is presented, along with innovative resource-sharing protocols that adeptly manage faults while ensuring real-time guarantees. The content satisfies Success Criteria 1 and 2 (SC-1 and SC-2) and addresses the Limitations 1, 2, and 3 highlighted in Chapter 2.

Chapter 4: This chapter presents the proposed RCM, which estimates the degree of competition for resources among tasks. Allocation strategies based on the RCM are also discussed, demonstrating the thesis's alignment with SC-3 and SC-4, address the Limitations 4, 5 and 6 that pointed out in Chapter 2.

Chapter 5: An RTA with novel techniques for DAG tasks is showcased. These techniques harness node-level parallelism and exclude non-contributory workloads when determining intra-task and inter-task delays. The content validates SC-5 and resolves the Limitations 7 and 8 that pointed out in Chapter 2.

Chapter 6: The thesis concludes by summarizing its main points, revisiting its contributions and central hypothesis, and suggesting potential avenues for future research.

Chapter 2

Review of Existing Literature

This literature review explores the foundational concepts of real-time systems, covering fundamental definitions, task models, and scheduling mechanisms, with an emphasis on Response Time Analysis (RTA).

We then discuss shared resources in real-time systems, introducing basic resource models and lock-based mechanisms essential for data integrity. Resource-sharing protocols, which offer guidelines for task interactions, are also explored, along with the evolution of RTA in this context.

Our focus shifts to Mixed-Criticality Systems (MCS), detailing task definitions within MCS and the evolution of RTA. We also touch upon conventional fault-tolerance approaches in embedded systems, leading to a holistic examination of fault-tolerance, resource-sharing, and MCS. The limitations of current research are addressed which set the stage for Chapter 3. The review further dissects task allocation algorithms, both traditional and those considering shared resources. A gap in resource-aware task allocation is highlighted, addressed in Chapter 4. We also explore the task model represented as Directed Acyclic Graphs (DAG), clarifying its core concepts and current research directions. The RTA of DAG tasks, especially in relation to RTA with intra-task priority assignment, is detailed, leading to our contributions in Chapter 5.

The review's structure is as follows: Section 2.1 offers an overview of real-time systems research. Section 2.2 focuses on shared resources and protocols. Section 2.3 covers MCS and fault-tolerance. Section 2.4 discusses task allocation challenges, and Section 2.5 provides an in-depth analysis of DAG tasks.

2.1 General Definitions of Real-Time Systems

In this section, we explore the core concepts of real-time systems, focusing on their definition, task models, and scheduling schemes. We will examine the intricacies of task scheduling in multiprocessor environments and conclude with an analysis of schedulability to ensure system reliability and efficiency. This overview will provide a foundational understanding of the critical components and operations in real-time systems.

2.1.1 Real-Time Task and System Models

Real-time systems, chiefly distinguished by their stringent timing requirements, play an integral role in numerous critical applications, ranging from autonomous vehicles to avionics systems [23]. These systems are not just defined by their processing speed but more importantly, by their determinism in fulfilling time-based commitments [63].

Real-Time Task Models

A real-time system typically contains a set of tasks Γ, in which the i^{th} task is denoted as τ_i. A task τ_i often contains a series of recurrent job instances $\{j_1, j_2, \ldots, j_n\}$. Each task τ_i can generally be defined by the following tuples: $\{T_i, D_i, Pri(\tau_i), P(\tau_i), C_{\tau_i}, U_{\tau_i}\}$.

- T_i represents the period of a task in real-time systems, referring to the time interval between consecutive instances of that task.
- D_i denotes the deadline by which each task must be completed.
- $Pri(\tau_i)$ indicates the priority of τ_i which is the ranking assigned to a task in an operating system, determining its execution order relative to other tasks. In this thesis, a larger number indicates a higher priority.

- $P(\tau_i)$ signifies the core on which the task is allocated.
- C_{τ_i} represents the pure Worst-Case Execution Time (WCET) of a task. This is the time the task computes on a core without considering interferences, blocking, or any other delays.
- U_{τ_i} is the utilization of the task τ_i, defined as the ratio of its computation time C_{τ_i} to its period T_i. Mathematically, $U_{\tau_i} = \frac{C_{\tau_i}}{T_i}$.

Based on their release patterns or periods tasks can be categorized into periodic, aperiodic, and sporadic tasks [23].

Periodic tasks, as the name implies, recur at constant time intervals. Every instance of such a task, or a *job*, carries a deadline that it must meet. For instance, the control loop in an autopilot system is a periodic task, which consistently reads sensor data and adjusts controls based on the readings.

In contrast, *aperiodic tasks* do not follow a fixed pattern and can occur spontaneously. A user command in a computer system, such as a mouse click or a keyboard press, serves as an example of an aperiodic task.

Finally, *sporadic tasks*, while resembling aperiodic tasks in their irregular occurrence, differ in that they enforce a minimum separation requirement between consecutive jobs and often come with hard deadlines. A safety monitor in a factory provides an apt example, requiring checks on machinery status following unpredictable events, with a requisite minimum time lapse between checks to avoid system overloading.

In addition, real-time tasks can be further classified as implicit-deadline, constrained-deadline, or arbitrary-deadline tasks, based on the relationship between a task's period and its deadline [68].

Implicit-deadline tasks have deadlines that coincide with the period. In mathematical terms, for each task i, this relationship can be expressed as: $D_i = T_i$.

These tasks are frequently encountered in periodic systems, such as control loops in embedded systems.

Constrained-deadline tasks have deadlines that are either equal to or less than the period. Mathematically, for each task i, this relationship is expressed as: $D_i \leq T_i$.

Conversely, *arbitrary-deadline tasks* may have deadlines that are shorter, equal to, or longer than the period. For these tasks, the mathematical relationship for each task with index i can be expressed as: $D_i \leq T_i$ or $D_i > T_i$, and are typically associated with complex, event-driven systems such as modern aviation control systems.

Real-Time System Models

As described by [68], real-time systems are further classified into hard and soft real-time systems based on the criticality of adhering to these deadlines. Hard real-time systems demand strict adherence to task deadlines, with any deviation potentially leading to catastrophic consequences. An exemplary instance is an air traffic control system, which must process radar data and relay updated aircraft positions to air traffic controllers within stringent time frames. Any lapse could result in dire events, such as mid-air collisions.

On the other hand, soft real-time systems offer a degree of leniency, where occasional missed deadlines might lead to a decrease in performance but remain within acceptable bounds. The in-flight entertainment system in aircraft serves as an example; while prompt responses enhance the passenger experience, minor delays will not have grave repercussions.

In the computing domain, the architecture of the processor plays a pivotal role in determining system performance and efficiency. Two primary architectures have emerged: homogeneous and heterogeneous [43].

Table 1: Table of Notations for Real-Time Tasks and Models

Notation	**Description**
Γ	Set of tasks in a real-time system
τ_i	i-th task in the set Γ
T_i	Period of task τ_i
D_i	Deadline of task τ_i
$Pri(\tau_i)$	Priority of task τ_i
$P(\tau_i)$	Core on which task τ_i is allocated
C_{τ_i}	WCET of task τ_i
U_{τ_i}	Utilization of task τ_i
Λ	Set of processors in a multiprocessor architecture
m	Number of cores in the system
P_a	a-th processor in the set Λ
$\Gamma(P_a)$	Set of tasks allocated to core P_a

Homogeneous Architecture: In this setup, all cores are functionally and performance-wise identical. They are capable of executing the same set of instructions, facilitating a streamlined programming and task scheduling environment. Such architectures are commonly found in traditional multi-core processors.

Heterogeneous Architecture: This design integrates diverse core types within a single system. For instance, a system might incorporate high-performance cores for demanding tasks alongside energy-efficient cores for less intensive operations. This configuration provides versatility, balancing performance and energy consumption.

Tasks are assumed to be scheduled on a multiprocessor architecture represented by a set of processors, Λ, and the number of cores in the system is denoted

as m (the terms "core" and "processor" are used interchangeably in this thesis). The a^{th} processor in Λ is denoted as P_a. $\Gamma(P_a)$ returns the set of tasks allocated to core P_a. While most parts of this thesis centers on the homogeneous architecture, we also transition our evaluation to a basic heterogeneous setup in Chapter 4, where identical cores operate at varying frequencies during runtime. Necessary notations for task and system models are summarized in Table 1.

2.1.2 Task Scheduling Mechanisms

Task scheduling is fundamental in real-time systems, dictating the order and timing of task executions. It guarantees timely and efficient task execution, ensuring the system adheres to its stringent time constraints.

Preemptive vs. Non-preemptive Scheduling

Task scheduling is central to a system's efficiency and responsiveness. Two key methodologies in this domain are preemptive and non-preemptive scheduling [30].

- **Preemptive Scheduling**: In this type of scheduling, the currently executing task can be interrupted (preempted) by a higher-priority task. Once the higher-priority task completes its execution or is blocked, the preempted task can resume its execution. This approach ensures that high-priority tasks are executed as soon as they become available, making it suitable for real-time systems where timely task execution is crucial. However it inevitably cause more frequent context switches.

- **Non-preemptive Scheduling**: Once a task starts its execution, it runs to completion without being interrupted. Even if a higher-priority task becomes available, it has to wait until the currently executing task completes. This approach can lead to longer waiting times for high-priority tasks but

ensures that once a task starts, it will not be interrupted, which can be beneficial in scenarios where task interruptions can lead to complications or inefficiencies.

Fixed-Priority and Dynamic Scheduling

Task scheduling can further be categorized into Fixed-Priority Scheduling (FPS) and dynamic scheduling. FPS, characterized by its fixed task execution order, brings predictability to the system. In contrast, dynamic scheduling adapts task priorities in real-time, offering the system flexibility to respond to changing conditions.

FPS: In this thesis, we focus on FPS, a scheduling scheme that is one of the most commonly employed in real-time systems due to its compelling features. Under FPS, tasks are assigned static priorities at the initial phase of system configuration. These priorities, which remain constant throughout the tasks' lifecycle, dictate the order in which tasks are selected for execution by the scheduler [68].

One of the principal advantages of FPS is the predictability it imparts to the scheduling process [30]. This predictability is invaluable in real-time environments, where guaranteeing the timely execution of tasks is paramount. It simplifies the system analysis, as the behavior of tasks under a set of known conditions can be assessed deterministically. This, in turn, makes both operations and system management more straightforward and robust.

Additionally, FPS is known for its relatively lower runtime overhead. Without the need for frequent, potentially complex recalculations of task priorities, the system can execute tasks efficiently, which is a crucial characteristic for systems where computational resources are at a premium [33].

Within the FPS paradigm, several specific algorithms exist for priority as-

signment:

- **Rate-Monotonic Priority Assignment (RMPA)**: RMPA assigns priority to tasks based on their period, with shorter periods indicating higher priority. This method is particularly useful in systems that handle periodic tasks, as it prioritizes tasks that demand more frequent attention [67].

- **Deadline-Monotonic Priority Assignment (DMPA)**: DMPA is a variation of RMPA optimized for systems where task periods and deadlines differ. DMPA prioritizes tasks with shorter relative deadlines, superseding the shortest period criteria [64].

- **Audsley's Optimal Priority Assignment (AOPA)**: AOPA is a sophisticated algorithm for optimal priority assignment under static-priority scheduling. It operates iteratively, assigning the lowest priority to the feasible task at that level, thereby excluding it from future consideration [4].

Dynamic Scheduling: Contrary to FPS methods, dynamic strategies decide task priorities at runtime. These priorities are subject to change during the system operation depending on various factors, such as task deadlines, task arrival times, or remaining computation time. These strategies generally manage aperiodic and sporadic tasks more effectively as they adapt to the evolving state of the system.

- **Earliest Deadline First (EDF)**: The EDF scheme is a dynamic scheduling strategy where the task with the nearest deadline is given the highest priority [23]. Specifically, EDF is proven to be optimal for uniprocessor systems, meaning that if a set of tasks with arbitrary arrival times is schedulable (i.e., all tasks can meet their deadlines) on a uniprocessor system under any scheduling algorithm, then that task set is guaranteed to be schedula-

ble under EDF. This optimality holds as long as the total CPU utilization of all tasks in the system is less than or equal to 1 (100%) [68].

- **Least Laxity First (LLF)**: Similar to EDF, LLF is a dynamic scheduling strategy that gives priority to tasks based on their laxity or slack time, assigning the highest priority to the task with the least remaining time after its execution time is subtracted from its deadline. This strategy is particularly beneficial in systems with variable execution times to prevent tasks from missing their deadlines [61].

However, dynamic scheduling can lead to increased overhead due to frequent task switching as priorities change with approaching deadlines. Handling these shifting priorities can be challenging, especially in systems with numerous tasks or variable behavior. While EDF works well for single-processor systems, its effectiveness can differ in multi-processor systems, which may require added synchronization [6].

Scheduling Paradigms in Multiprocessor Systems

Over the years, various scheduling methods have been developed to efficiently manage task execution within real-time multiprocessor systems, each offering unique advantages [7, 69, 71].

- **Fully-Partitioned Scheduling [69]:** Tasks are assigned to specific processors and remain there throughout their lifecycle. This approach is predictable and reduces overhead due to the absence of inter-processor communication for task migrations. However, it demands careful task allocation to avoid imbalances in processor utilization.

- **Semi-Partitioned Scheduling [71]:** While tasks are primarily assigned

to individual processors, certain tasks can migrate between processors, providing better flexibility and potentially improved processor utilization.

- **Global Scheduling [65]:** Tasks are placed in a system-wide queue and can be executed by any available processor. This approach offers flexibility but may introduce overhead due to frequent task migrations.

- **Global List Scheduling [76]:** A specific type of global scheduling where tasks are organized in a priority list. Processors pick tasks based on their priority, ensuring that high-priority tasks are executed first.

- **Federated Scheduling [7]:** A hybrid approach that combines partitioning and global scheduling. Larger tasks are partitioned onto specific processors, while smaller tasks are globally scheduled, optimizing resource utilization.

The selection among these strategies depends on the specific requirements of the system being designed, underscoring the significance of a deep understanding of each method.

2.1.3 Schedulability Analysis

In the sphere of real-time systems, the reliable execution of tasks within their stipulated deadlines is of paramount importance. To certify this essential attribute, a robust approach known as schedulability analysis [30], typically enacted through schedulability tests, is employed. These tests represent sophisticated analytical procedures designed to ascertain whether a particular set of tasks, under a predetermined scheduling policy, can invariably fulfill their deadlines.

The types of schedulability tests is multi-faceted, with a principal categorization based on the form of assurance they furnish [29]: *necessary*, *sufficient*,

and *exact* (or optimal) conditions. A necessary condition, if left unsatisfied, unequivocally confirms the non-schedulability of a task set. Conversely, even upon meeting this condition, a task set's schedulability cannot be confirmed, necessitating further analysis. Sufficient conditions, if met, provide the assurance of a task set's schedulability. However, an inability to satisfy this condition does not conclusively pronounce the task set as unschedulable. An exact or optimal condition, serving as both a necessary and sufficient determinant, delivers a definitive verdict: a task set that satisfies this condition is categorically schedulable, while those failing to meet this criterion are deemed non-schedulable.

The utilization test serves as a foundational test for determining the schedulability of tasks in real-time systems. However, its specific criteria vary across different scheduling algorithms. In *sufficient* scenarios, fixed-priority scheduling algorithms, such as RMPA and DMPA, assign tasks a static priority that remains unchanged over time. The derived schedulability tests for these algorithms, particularly for periodic tasks, are rooted in worst-case scenarios. Specifically, the formula $n(2^{1/n} - 1)$, as proposed by Liu and Layland [67], establishes an upper bound on CPU utilization for a set of n tasks. This threshold, which is notably less than 100%, accounts for the worst-case interference from higher-priority tasks that can potentially delay the execution of those ranked lower. As the task count escalates, this bound asymptotically approaches a value below 100%, approximately 69.3%, ensuring that all tasks can meet their deadlines even in the most adverse conditions.

Conversely, dynamic scheduling algorithms, exemplified by the EDF method, exhibit a more fluid approach, adjusting task priorities based on specific criteria, such as impending deadlines. The 100% utilization criterion for EDF suggests that if tasks collectively do not demand more than the available CPU time, they are feasibly schedulable. This is attributed to the algorithm's strategy of always

prioritizing the task with the most imminent deadline. However, a caveat exists: while a CPU utilization below 100% does not universally guarantee schedulability due to variables like task phasing, surpassing this threshold unequivocally renders a task set non-schedulable. In this case, utilization test is deemed as *necessary* test.

More intricate scenarios, characterized by task dependencies, blocking times, and resource sharing, necessitate the formulation of advanced schedulability tests. Often grounded in RTA, these tests primarily are *sufficient* tests which focus on computing a task's Worst-Case Response Time (WCRT) and comparing it to its designated deadline [13].

In systems devoid of shared resources, a series of tasks exists, each with a distinct priority. The RTA for a task τ_i is exemplified. As depicted in Equation (1), R_i denotes the WCRT of τ_i.

$$R_i = C_{\tau_i} + B_i + \sum_{\tau_h \in \text{lhp}(i)} \left\lceil \frac{R_i}{T_h} \right\rceil C_{\tau_h} \tag{1}$$

More specifically, C_{τ_i} represents the WCET of τ_i. Without shared resources, tasks might be blocked due to non-preemptive (NP) segments run by the underlying RTOS. This halts certain tasks as the system operates within an NP segment. Since such blocking can occur only once during the release of a real-time task, the blocking duration can be directly bounded by the longest non-preemptive segment in the RTOS, as shown in Equation (2). Here, B_i signifies the maximum duration of the RTOS's NP segments. In $\sum_{\tau_h \in \text{lhp}(i)} \left\lceil \frac{R_i}{T_h} \right\rceil C_{\tau_h}$, $\text{lhp}(i)$ denotes the set of tasks on the same processor with a priority higher than τ_i, and $\frac{R_i}{T_h}$ represents the number of times the highe-priority task τ_h can be released during the period R_i. The overall notation indicates the total interference incurred by τ_i from tasks on the same processor that have a higher priority [21].

$$B_i = \max(NP_1, NP_2, \ldots, NP_n) \tag{2}$$

In situations where an analytical approach is deemed impractical due to system intricacies, simulation-based testing methods emerge as a practical alternative. These methods involve simulating the system's function over a set duration, often a hyperperiod, followed by identifying potential deadline breaches [16].

To sum up, schedulability tests are a cornerstone in the domain of real-time systems. They provide essential guarantees of temporal accuracy, a trait crucial in many applications, especially those with strict safety or performance requirements. This field remains an active area of research, with novel methods continually emerging to address the challenges of increasingly intricate system models.

2.1.4 Summary

In this section, we provide an overview of real-time systems, emphasizing their defining attributes, task models, and the mechanisms that govern their operation. Central to our discussion is the FPS method, a prevalent approach in real-time systems due to its predictability and efficiency. We also spotlight the RTA, a critical tool for assessing task schedulability and ensuring that tasks meet their deadlines. Through this exploration, we aim to offer a foundational understanding of the key components and methodologies that underpin the reliable functioning of real-time systems.

2.2 Resource Sharing in Real-Time Systems

In this section, we introduce the shared resources model central to our thesis, outlining the specific resources we focus on. We will examine lock-based mechanisms that govern access to these resources, ensuring orderly task execu-

tion. Additionally, we will discuss the protocols that regulate task interactions with shared resources, a crucial step in maintaining system integrity. Lastly, we will undertake a schedulability analysis incorporating shared resources and the applied protocols, aiming to assess a system's ability to meet task deadlines effectively. This discussion lays a robust groundwork for further explorations in this thesis.

2.2.1 Shared Resources

Shared resources play a pivotal role in the realm of computer science and programming, especially in real-time systems where tasks have strict timing constraints [110]. These resources include data structures, such as arrays, lists, and trees, which are systematic ways of organizing and storing data. They also consist of variables, essential memory storage locations that hold data values, and specialized memory spaces like buffers or caches tailored for specific purposes. These spaces can speed up data access or provide intermediary storage.

In real-time systems, managing concurrent access to shared resources is challenging. When multiple tasks or processes need simultaneous access to a resource, they enter a *critical section* of the code. It is essential to ensure mutual exclusivity within this critical section, allowing only one task to access the shared resource at a time.

To guarantee safe and efficient concurrent access, synchronization mechanisms like locks, semaphores, or critical section protocols are used. But in real-time contexts, these mechanisms must be carefully designed to ensure tasks do not miss their deadlines while waiting for resource access [88].

In this thesis, we explore the intricacies of shared resources in multicore real-time systems. We posit that within the system, there exists a set of resources, denoted as $\mathbb{R}$. Every resource in this set can be accessed by all tasks, but only

in a mutually exclusive manner. This exclusivity is achieved by executing the associated *critical section*. A shared resource with index x, represented as r^x, is defined by two notations: c^x and N_i^x. Here, c^x denotes the computation time required to access the shared resource r^x. In contrast, N_i^x quantifies the number of requests from τ_i within a single release. The function $F(\cdot)$ specifies the set of resources requested by the given tasks.

In addition, due to the presence of shared resources in the system, the total computation time for τ_i (denoted as $\widehat{C}_{\tau_i}$) must also account for the resource computation time. This is given by

$$\widehat{C}_{\tau_i} = C_{\tau_i} + \sum_{r^x \in F(\tau_i)} N_i^x \cdot c^x \tag{3}$$

where C_{τ_i} denotes the pure WCET of τ_i when executed on a processor without accessing shared resources. The term

$$\sum_{r^x \in F(\tau_i)} N_i^x \cdot c^x \tag{4}$$

represents the total time τ_i spends accessing shared resources. Here, $F(\tau_i)$ returns the set of resources requested by τ_i, and $N_i^x \cdot c^x$ computes the total time τ_i spends accessing resource r^x. The utilization of a task, U_{τ_i}, is subsequently determined by

$$U_{\tau_i} = \frac{\widehat{C}_{\tau_i}}{T_i} = \frac{C_{\tau_i} + \sum_{r^x \in F(\tau_i)} N_i^x \cdot c^x}{T_i} \tag{5}$$

which is the total computation time divided by the task's period. For clarity, the notations of shared resources are provided in Table 2 below.

2.2.2 Lock-Based Mechanisms

In the field of real-time systems, ensuring data integrity and the predictability of task behaviors is paramount. One prevalent method for safeguarding data and mitigating race conditions is the lock-based synchronization technique. This

Table 2: Notations of Shared Resources

Symbol	Meaning
$\mathbb{R}$	Set of shared resources in the system
r^x	A shared resource with index x
c^x	Computation time to access the resource r^x
N_i^x	Number of requests from τ_i to resource r^x within a single release
$F(\cdot)$	Function specifying the set of resources requested by a task
$\widehat{C}_i$	Total computation time for τ_i considering shared resources

approach works harmoniously alongside non-blocking strategies to create a cohesive synchronization system. In the lock-based paradigm, specific locks guard critical sections. Tasks can only enter these sections upon acquiring the relevant lock. When one task holds a lock, others must patiently await its release.

This strategy encompasses several locking primitives, including mutex locks, semaphores, and monitors [25]. Based on a task's behavior during its waiting period, these primitives can be classified as [104]:

1. **Suspension-based Locks:** Under this mechanism, tasks wanting an engaged lock will temporarily step back, yielding the processor. They transition to an idle state and join a priority queue. Upon the lock's release, the task at the queue's forefront, usually with the highest priority, resumes and claims the lock.

2. **Spin Locks:** Contrary to the suspension-based locks, tasks with spin locks stay active, even if the required lock is unavailable. They remain in a *busy-wait* state, repeatedly checking the lock's status. Though often governed by a First-in-First-out (FIFO) methodology, a priority-based system can also serve them. This type of locking is particularly favored in kernel-level

operations.

However, the lock-based approach is not devoid of challenges [77]. *Deadlocks* present a significant concern. In such scenarios, tasks, intertwined in a nested quest for resources, reach a standstill due to mutual dependencies. Additionally, there's the menace of livelocks, where tasks continuously defer lock acquisition to each other, achieving no progress. Real-time systems counteract these pitfalls by introducing additional rules such as not allowing a task to hold resources while waiting for others to avoid *deadlocks* and set strict request order to avoid *livelocks*. It is important to note that this thesis does not consider nested resource sharing. In other words, a task can interact with only one resource at any given time. However, this limitation can be circumvented using group locks, as discussed in [109].

Nevertheless, the incorporation of locks, while indispensable, introduces its own set of complexities. One prominent challenge is priority inversion, which could significantly disrupt the predictable functioning of a real-time system. When a high-priority task finds itself stalling due to a lower-priority task's operations, unpredictability ensues, risking missed task deadlines. For clarity, consider the following illustrative example:

Example 1.

τ_1 : *A task with high priority.*

τ_2 : *Holding a priority lower than* τ_1.

r^1 : *A resource desired by* τ_2.

Based on the tasks and resource setup, Figure 1 presents an execution scenario that demontrates what priority inversion is.

- τ_2 *initiates its operation at time point* $t = 0$ *and acquires* r^1 *, then remains non-preemptive from* $t = 2$ *to* $t = 4$.

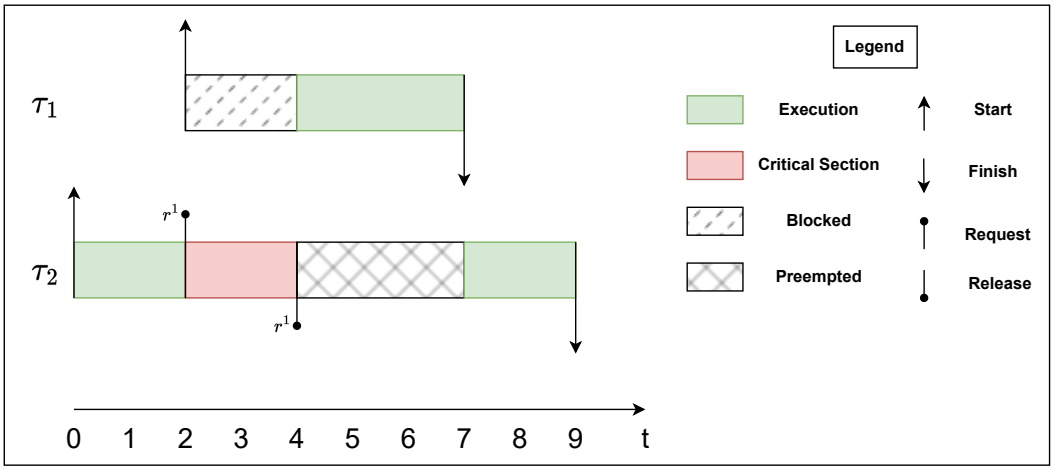


Figure 1: The execution chart for priority inversion

- *During τ_2's execution with r^1 , τ_1 readies for execution at $t = 2$, and aims to execute on the processor.*
- *Given that τ_2 currently possesses r^1, τ_1, despite its higher priority, is blocked from $t = 2$ to $t = 4$.*
- *This situation, wherein τ_1 is hindered by τ_2 due to a shared resource access, epitomizes priority inversion.*
- *After τ_2 releases r^1 at $t = 4$, τ_1 preempts it and executes until $t = 7$*
- *τ_2 resumes its execution at $t = 7$ and finishes executing at $t = 9$ in this example*

The presented scenario highlights the behaviors of priority inversion. It represents a divergence from the expected sequence of task execution based on priority. When a high-priority task is compelled to stand by because a low-priority task accesses a shared resource, the system's predictable feature faces disruption. Such a disturbance can lead to missed deadlines, especially for pivotal high-priority tasks.

Completely eliminating priority inversion may appear as an aspirational target, but it is not always the primary aim. The objective is to limit the duration and effects of these inversions. In doing so, real-time systems can ensure all tasks to meet their deadlines.

2.2.3 Resource Sharing Protocols

The pivotal role of shared resources in fostering concurrent task interactions highlights the imperative for dependable protocols. These protocols are crucial not only to ensure smooth access but also to prevent race conditions, safeguard data integrity, and maintain the system's overall dependability. Historically, the field of resource sharing in single-core and multicore real-time systems has benefited from comprehensive research, leading to the development of numerous resource sharing protocols [2, 22, 41]. For an in-depth review of these protocols can be found in survey presented in [15].

Protocols for Uniprocessor Systems

In single-processor systems, the Priority Ceiling Protocol (PCP) is widely used to control how tasks access shared resources. Two main adaptations of PCP are well-documented in [81]: the Original Priority Ceiling Protocol (OPCP) and the Immediate Priority Ceiling Protocol (IPCP).

In the OPCP, there are two distinct ceiling priorities: the resource ceiling and the system ceiling. The resource ceiling priority is set to the priority of the highest-priority task that may request the resource. Meanwhile, the system ceiling priority reflects the highest resource ceiling priority of any resource currently in use. With OPCP in place, a task can only acquire a lock if its priority is higher the current system ceiling. If not, the task will be blocked until it can meet this condition.

With the IPCP, each shared resource has an assigned ceiling priority, which is determined by the highest priority of tasks that might request the resource. Within the IPCP protocol, when a task secures a shared resource, its priority is temporarily elevated to match the resource's ceiling priority. This elevation in priority ensures that, while in possession of a shared resource, the task remains immune to preemption by other tasks that have priorities lower than the ceiling. An execution example of PCPs is demonstrated below.

Example 2. *Consider three tasks and their associated priorities:*

$$\tau_1 : \textit{Priority 3 (highest)}$$

$$\tau_2 : \textit{Priority 2}$$

$$\tau_3 : \textit{Priority 1 (lowest)}$$

We also have the following resources with their respective resource ceilings:

$$r^1 : \textit{Ceiling Priority 2 (accessed by } \tau_2 \textit{ and } \tau_3\textit{)}$$

$$r^2 : \textit{Ceiling Priority 3 (accessed by } \tau_1\textit{)}$$

Given this system setup, with assistance of Figure 2, we demonstrate an execution scenario of tasks under OPCP:

- τ_3 *starts at* $t = 0$*, requests* r^1 *at* $t = 2$ *and successfully locks the resource. The system ceiling is then raised to 2.*
- τ_2 *starts and requests* r^1 *at* $t = 2$*, but blocked by* τ_3 *as its priority 2 is not higher than the current system ceiling.*
- *Subsequently,* τ_1 *starts and requests* r^2 *at* $t = 4$*. As its priority 3 is higher than the current system ceiling, it preempts* τ_3 *and lock* r^2 *successfully.*
- τ_3 *resumes its execution at* $t = 6$ *as* τ_1 *releases the resource and finishes executing, then it releases* r^1 *and finishes executing at* $t = 7$*.*

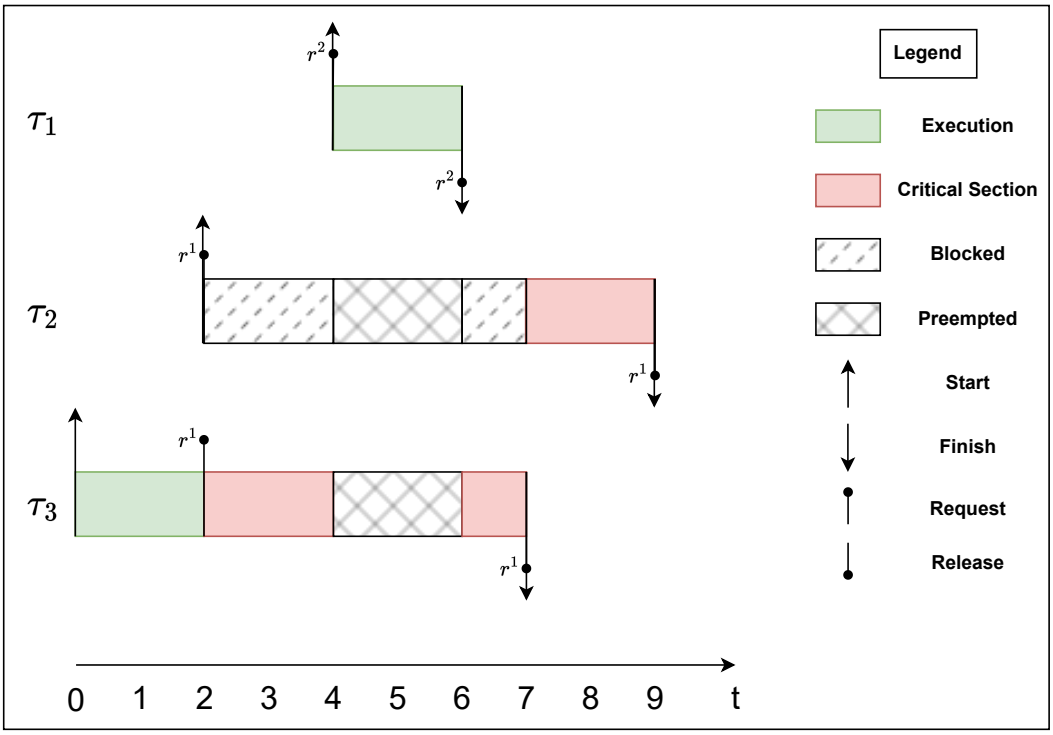


Figure 2: The execution chart for tasks under PCPs

- *In the mean time, τ_2 finally secures r^1 at $t = 7$ and finishes executing at $t = 9$.*

With the same system setup, the task behaviors under IPCP is similar to OPCP and can so be described by referencing Figure 2.

- *τ_3 starts at $t = 0$ and requests r^1 at $t = 2$ and successfully lock the resource. Its priority is immediately boosted to 2.*

- *τ_2 starts and requests r^1 at $t = 2$, but blocked by τ_3 as its priority 2 is not higher than the boosted priority of τ_3.*

- *Subsequently, τ_1 starts and requests r^2 at $t = 4$. As its priority 3 is higher than τ_3's, it preempts τ_3 and locks r^2 successfully with unchanged priority (its priority is equal to the ceiling priority of r^2).*

- *τ_3 resumes its execution still with boosted priority at $t = 6$ as τ_1 releases the resource and finishes its execution. At $t = 7$, τ_3 releases the resource, retrieves its base priority and exists the system*

- *τ_2 finally secures r^1 at $t = 7$ and exists the system at $t = 9$.*

The primary differentiation between IPCP and OPCP lies in the dynamic elevation of priority inherent to IPCP. While protocols like OPCP inhibit tasks based on the prevailing system ceiling, IPCP uniquely elevates the priority of the task in question. This ensures that the task remains impervious to preemption by tasks of intermediary priorities that do not necessitate the locked resource.

Resource-Sharing Protocols for Multiprocessor Systems

In the multiprocessor aspect, out of the many existing protocols, the Multiprocessor Stack Resource Protocol (MSRP) [41] merits special attention. Specifically crafted for fully-partitioned systems, MSRP is a FIFO spin-based resource-sharing protocol. Under MSRP, each globally shared resource is linked with a FIFO queue. Tasks requesting such a resource are added to the queue and busy-wait (spin) non-preemptively until they reach the queue's head, upon which the resource is assigned to them. Once a task is granted a resource, it carries on its non-preemptive execution until the resource is relinquished. In contrast, when dealing with a local resource (i.e., a resource shared within one core), the IPCP is employed. The MSRP is examplified below.

Example 3. *Consider a two-core computer system, P_1 and P_2, dedicated to managing tasks in a computational environment.*

Tasks and Resources:

τ_1 *on* P_1 : *Requires* r^1.

τ_2 *on* P_2 : *Requires both* r^1 *and* r^2.

τ_3 *on* P_2 : *Requires* r^2.

r^1 : *Globally shared computational module.*

r^2 *on* P_2 : *Local computational unit exclusive to* P_2.

Scenario 1 - Accessing the Global Resource r^1*:*

- τ_1 *on* P_1 *initiates its operation and locks* r^1.
- *Simultaneously,* τ_2 *on* P_2 *requires* r^1. *Since* τ_1 *is currently using the resource,* τ_2 *enters the FIFO queue and spins on* P_2 *non-preemptively.*
- *After* τ_1 *releases* r^1, τ_2 *becomes the head of the FIFO queue and starts accessing* r^1 *non-preemptively on* P_2.
- *Concurrently, if* τ_3 *is released, it will be blocked by* τ_2 *from computing on* P_2*, regardless of its priority.*

Scenario 2 - Accessing the Local Resource r^2*:*

- τ_2 *wants to access* r^2. *Due to the priority ceiling protocol, its priority is elevated.*
- τ_3 *is subsequently blocked from executing until* τ_2 *finishes its operation regardless of its priority.*

In this structure, MSRP ensures efficient access to both global and local resources, reducing delay for low-priority tasks and ensuring smooth task executions.

2.2.4 RTA for Shared Resources

The implementation of resource-sharing protocols can prevent race conditions among tasks in the system which in turn safeguards data integrity. However, the timely behavior of tasks, especially during resource sharing, can lead to additional waiting times or priority inversion—termed as *blocking*—owing to the mutually exclusive nature of resource access. Capturing the intricacies of these protocols accurately is vital to ensure the timing correctness of tasks.

The blocking effect is primarily divided into *spin delay* and *arrival blocking* [109]. *Spin delay* can further be categorized into *Direct* and *Indirect spin delay.*

Direct spin delay occurs when a task is directly obstructed from accessing a shared resource due to other concurrent remote accesses. For instance, under MSRP, when task τ_1 requests a resource r^1, tasks on other processors, say τ_2 and τ_3, are ahead of τ_1 in the FIFO queue because they requested r^1 earlier. The time τ_1 spends waiting for r^1 before it can lock it is referred to as direct spin delay.

In contrast, *indirect spin delay* arises when a task is preempted by a higher-priority local task that is itself directly blocked. For example, under MSRP, when τ_1 is executing on the processor normally without accessing any shared resource, a higher-priority task τ_2 preempts τ_1 to compute on the processor and requests for a global shared resource. Here, τ_1 can only resume its computation when τ_2 finishes its execution. Therefore, the *spin delay* experienced by τ_2 will transitively cause a delay to τ_1, termed as *indirect spin delay.*

Arrival blocking, on the other hand, transpires when a newly released high-priority task, like τ_2, is immediately blocked by a lower-priority task, τ_1. This can be because τ_1 is either currently requesting or locking a shared resource. As a result, it remains non-preemptive under MSRP until it completes its operation

and releases the lock. Thus, a higher-priority task like τ_2 cannot preempt τ_1 directly, leading to a delay known as *arrival blocking*.

Original RTA for MSRP

The RTA designed to capture the system's worst-case timely behavior under fully-partitioned systems scheduled by preemptive FPS with the application of MSRP was first developed by [41]. As shown in Equation (6), the analysis aims to calculate the WCRT of each task, denoted as R_i, which comprises the total WCET ($\widehat{C}_i$), the arrival blocking (B_i), and the interference imposed by the set of local tasks with a higher priority than τ_i, represented as $\sum_{\tau_h \in \text{lhp}(i)} \left\lceil \frac{R_i}{T_h} \right\rceil \cdot \widehat{C}_h$.

$$R_i = \widehat{C}_i + B_i + \sum_{\tau_h \in lhp(i)} \left\lceil \frac{R_i}{T_h} \right\rceil \cdot \widehat{C}_{\tau_h} \tag{6}$$

Further exploring R_i, as illustrated in Equation (7), the term $\widehat{C}_i$ comprises the pure WCET of τ_i without accessing any shared resources, denoted as C_{τ_i}, and the time spent accessing resources, given by $\sum_{r^x \in F(\tau_i)} N_i^x \cdot e^x$. Here, $F(\tau_i)$ represents the set of resources requested by τ_i, N_i^x indicates the number of requests that τ_i sends to r^x during a single release, and e^x denotes the total time spent on each access to r^x.

$$\widehat{C}_i = C_{\tau_i} + \sum_{r^x \in F(\tau_i)} N_i^x \cdot e^x \tag{7}$$

The amount of time that a task can spend accessing r^x once is illustrated in Equation 8. In this equation, the function $map(G(r^x))$ is utilized to identify the set of processors that contain tasks requesting access to r^x. The notation $||$ indicates the size of this set of processors, representing the maximum number of requests that can be sent to r^x concurrently. This analysis assumes a worst-case scenario wherein τ_i is perpetually placed at the end of the FIFO queue, forcing

it to wait until requests from other processors have been fully serviced. Consequently, e^x is determined by multiplying the maximum number of concurrent requests by the time needed to access r^x, denoted as c^x.

$$e^x = |map(G(r^x))| \cdot c^x \tag{8}$$

The arrival blocking is denoted by B_i, as illustrated in Equation (9). Here, $\hat{e}_i$ signifies the maximum arrival blocking that τ_i can experience, while $\hat{b}$ represents the non-preemptive section of the operating system, as previously discussed. The arrival blocking of τ_i takes place when τ_i is released but is hindered from execution by a local low-priority task (e.g., τ_{ll}). This hindrance arises when the latter is in the process of requesting or utilizing a shared resource and hence, cannot be preempted by τ_i.

$$B_i = max\{\hat{e}_i, \hat{b}\} \tag{9}$$

Equation (10) aims to discern the peak arrival blocking, which can also be perceived as the maximum time τ_{ll} can allocate to accessing a shared resource r^x, essentially the upper limit of e^x. As indicated in Equation (10), the resource should be sought by τ_{ll}; thus, the count of requests from τ_{ll} to r^x should be non-zero, or in mathematical terms, $N_{ll}^x > 0$.

Furthermore, the resource resulting in the blocking effects can be global, attributed to the MSRP's provision that τ_{ll} maintains its non-preemptive status during both request and execution phases. Conversely, the said resource might be local, boasting a ceiling priority surpassing that of τ_i. This allows τ_{ll} to perform under an elevated priority, in turn obstructing τ_i. This scenario is symbolized by the condition $Pri(r^x, P(\tau_i)) \geq Pri(\tau_i)$, where $Pri(r^x, P(\tau_i))$ reveals the priority ceiling of r^x when the resource is specific to the processor that τ_i is allocated $(P(\tau_i))$. Necessary notations are concluded in Table 3.

Table 3: Table of Notations for Original MSRP

Notation	Definition
$\widehat{C}_i$	Worst-case execution time of task τ_i including shared resource access time
C_{τ_i}	Pure computation time of task τ_i excluding shared resource access time
B_i	Arrival blocking time of task τ_i
$\text{lhp}(i)$	Set of local tasks with higher priority than τ_i
$F(\tau_i)$	Set of resources requested by task τ_i
e^x	Total time spent accessing resource r^x once
$map(G(r^x))$	Function to identify the set of processors that contain tasks requesting access to r^x
$\hat{e}_i$	Maximum arrival blocking that task τ_i can experience
τ_{ll}	A local low-priority task
N_{ll}^x	Number of requests from τ_{ll} to r^x
$Pri(r^x, P(\tau_i))$	Priority ceiling of resource r^x when local to processor $P(\tau_i)$

$$\hat{e}_i = max\{e^x | N_{ll}^x > 0 \wedge (r^x \; is \; global \vee Pri(r^x, P(\tau_i)) \geq Pri(\tau_i))\} \tag{10}$$

This brings us to the conclusion of the original RTA of MSRP, as illustrated in the seminal work by [41]. This early analysis operates under the assumption that each request originating from a local processor to r^x can be obstructed by a constant number of requests from other cores, i.e., $|map(G(r^x))|$, irrespective of the accounting for the exact number of requests from remote processors. Unfortunately, this assumption leads to the inclusion of many extra, repeated spin delays, resulting in a significantly overestimated WCRT bound.

Advanced RTA for MSRP

The evolution of schedulability analysis for MSRP took a significant leap forward with the work of [104], which notably improved the degree of pessimism in calculating the WCRT. As shown in Equation (11), the WCRT of task τ_i is systematically computed. In this framework, C_{τ_i} stands for the WCET of τ_i excluding any interactions with shared resources. On the other hand, E_i represents the cumulative time expended in accessing shared resources, accounting for both τ_i and the higher-priority tasks operative ($lhp(i)$) on the local processor.

The term $\sum_{\tau_h \in lhp(i)} \left\lceil \frac{R_i}{T_h} \right\rceil \cdot C_h$ encapsulates the interference introduced by all tasks in $lhp(i)$. Specifically, $\left\lceil \frac{R_i}{T_h} \right\rceil \cdot C_h$ conveys the interference from a higher-priority task τ_h. This concept aligns with that introduced in Equation 1, where $\left\lceil \frac{R_i}{T_h} \right\rceil$ calculates the release time of τ_h during τ_i's execution and then multiplies by τ_h's pure WCET.

$$R_i = C_{\tau_i} + E_i + B_i + \sum_{\tau_h \in lhp(i)} \left\lceil \frac{R_i}{T_h} \right\rceil \cdot C_h \tag{11}$$

The key component of the WCRT of MSRP analysis is the total resource accessing time, denoted as E_i. As illustrated in Equation (12), E_i iterates over all resources in the system $\mathbb{R}$ to identify the resource that could directly or indirectly prolong the response time of τ_i. The term N_i^x represents the number of requests from τ_i to r^x in a single release. Additionally, $Nh_i^x(R_i)$ stands for the number of requests sent from task in $lhp(i)$ during the period R_i. Meanwhile, $Nr^x(R_i, R_j)$ signifies the count of requests from all remote processors capable of obstructing requests from the local processor during the period R_i and a release jitter R_j, in other words, the requests that can induce *direct* or *indirect spin delays* to τ_i.

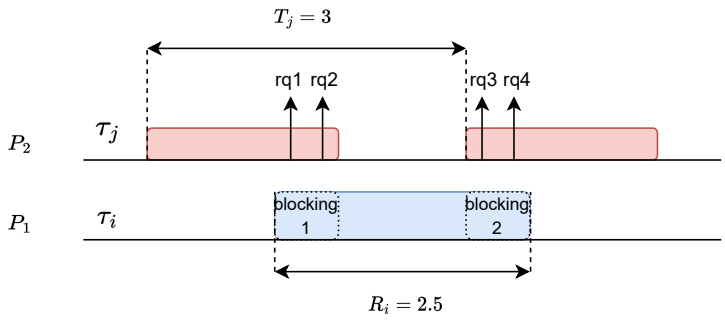


Figure 3: The back-to-back hit

$$E_i = \sum_{r^x \in \mathbb{R}} \left(N_i^x + Nh_i^x(R_i) + Nr^x(R_i, R_j) \right) \times c^x \tag{12}$$

$Nh_i^x(R_i)$ is demonstrated in Equation (13), where $\tau_h \in lhp(i)$. The expression $\left\lceil \frac{R_i}{T_h} \right\rceil$ calculates the number of times τ_h will be released, with N_h^x representing the number of times τ_h will request r^x in a single release.

$$Nh_i^x(R_i) = \sum_{\tau_h \in lhp(i)} \left\lceil \frac{R_i}{T_h} \right\rceil \cdot N_h^x \tag{13}$$

In the meantime, as illustrated in Equation (14), $Nr_a^x(R_i, R_j)$ denotes the number of requests for the resource r^x initiated by all tasks on processor P_a (denoted as $\Gamma(P_a)$), over the interval defined by the duration R_i and the jitter R_j. It is important to note that the inclusion of the jitter R_j accommodates the back-to-back hit phenomenon of resource requests [109]. Referencing Figure 3, the remote task τ_j requests the shared resource four times in two releases (ranging from rq1 to rq4). In this scenario, to accurately account for the number of requests that τ_j can send during τ_i's execution, the formula $\left\lceil \frac{R_i}{T_j} \right\rceil \times 2 = \lceil \frac{2.5}{3} \rceil \times 2 = 2$ proves insufficient. Consequently, a jitter represented by $\mu = R_j$ is introduced to incorporate the back-to-back hits from τ_j, exemplified as $\left\lceil \frac{R_i + R_j}{T_j} \right\rceil \times 2 = 2 \times 2$.

$$Nr_a^x(R_i, R_j) = \sum_{\tau_j \in \Gamma(P_a)} \left\lceil \frac{R_i + R_j}{T_j} \right\rceil \cdot N_j^x \tag{14}$$

Finally, $Nr^x(R_i, R_j)$ is utilized to denote the total number of requests on all remote processors that can impose *direct* or *indirect spin delay* on τ_i. The notation $P_a \notin P(\tau_i)$ indicates all processors excluding the one τ_i is currently utilizing. The function $\min\{\iota, \kappa\}$ is applied to determine the lesser value between ι and κ. Here, $\iota = N_i^x + Nh_i^x(R_i)$ and $\kappa = Nr_a^x(R_i, R_j)$, where ι represents the cumulative number of requests sent from the local processor, encompassing both the requests initiated by τ_i itself (N_i^x) and those initiated by tasks in $lhp(i)$, $Nh_i^x(R_i)$, during the period defined by R_i. The constraint applied here is that the maximum number of requests blocking the local requests is determined by the minimum of the two values, ι or κ. This is grounded on the principle that a processor can send only one request at a time, hence a request can be delayed by another request from a processor at most once. An example illustrating this analysis is provided in Example 4. All necessary notations are summarized in Table 4

$$Nr^x(R_i, R_j) = \sum_{P_a \notin P(\tau_i)} \min\left\{N_i^x + Nh_i^x(R_i), Nr_a^x(R_i, R_j)\right\} \tag{15}$$

Example 4. *Figure 4 provides an example that illustrates the total resource-access time that τ_i can incur due to accesses to a resource r^x. In this figure, during the release of τ_i, it executes on P_1 and requests the resource once, while tasks in $lhp(i)$ preempt τ_i and send two requests to the resource on core P_1. Tasks on core P_2 request the same resource five times during the same period. Meanwhile, the tasks on core P_3 request the resource only twice. As illustrated in the figure, the first request from P_1 can incur delays from one request from both P_2 and P_3, as shown in the queue labeled FIFO1. Similarly, the second request*

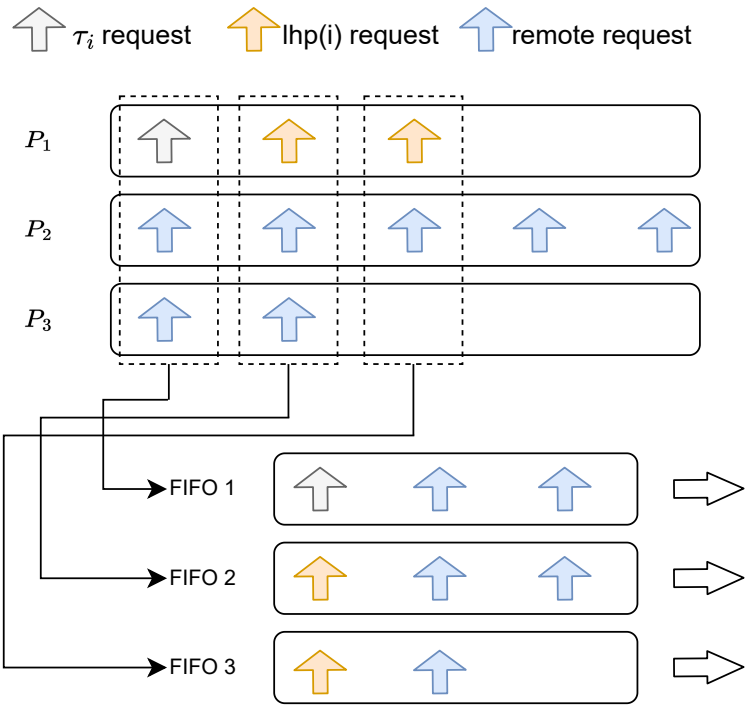


Figure 4: Resource contention analysis for one shared resource

from P_1 faces the same scenario, depicted in FIFO2. However, the third request from P_1 can only incur one delay from P_2, since tasks on P_3 only request twice in total. Summarizing, on P_2 a maximum of three requests can cause spin delay to τ_i; one of these delays affects the request from τ_i directly, while the other two delay the requests from $lhp(i)$, thereby delaying τ_i transitively. For core P_3, the tasks there request the resource only twice, meaning they can induce a direct or indirect delay to τ_i a maximum of two times. Consequently, τ_i will be blocked by remote requests five times, leading to a worst-case spin delay of seven occurrences.

The arrival blocking is denoted by B_i, as shown in Equation (16), where $\hat{e}_i$ represents the maximum arrival blocking that τ_i can incur, and $\hat{b}$ designates the non-preemptive section of the operating system, as introduced earlier.

$$B_i = max\{\hat{e}_i, \hat{b}\} \tag{16}$$

Table 4: Table of Notations for Advanced MSRP

Notation	**Description**
E_i	Total resource accessing time for task τ_i and lhp(i)
$Nh_i^x(R_i)$	Number of requests sent from local tasks with higher priority lhp(i) during the period R_i
$Nr_a^x(R_i, R_j)$	Number of requests from processor P_a that can cause direct or indirect spin delays during the periods R_i and R_j
$Nr^x(R_i, R_j)$	Number of requests from remote processors that can cause direct or indirect spin delays during the periods R_i and R_j
$\Gamma(P_a)$	Set of all tasks on processor P_a
$min(\iota, \kappa)$	Function to return the smaller value between ι and κ

The arrival blocking of τ_i occurs when τ_i is released and prevented from executing by a local low-priority task (e.g., τ_{ll}), which is either requesting or executing a shared resource and, therefore, cannot be preempted by τ_i. This scenario is further detailed in Equation (17), where the set of resources potentially causing arrival blocking to τ_i is denoted as $F^A(\tau_i)$. The total arrival blocking time is determined by multiplying the number of requests to r^x causing arrival blocking, represented as $|\alpha_i^x|$, by the duration of c^x. It is important to note that a task can only incur arrival blocking once; after τ_i initiates its execution, τ_{ll} cannot block it again. Consequently, the function max{} identifies the resource that can induce the worst-case arrival blocking.

$$\hat{e}_i = max\{|\alpha_i^x| \cdot c^x | r^x \in F^A(\tau_i)\} \tag{17}$$

The set of resources $F^A(\tau_i)$ is bounded in Equation (18). Firstly, N_{ll}^x indicates the number of requests from τ_{ll} to r^x, and this value must be greater than 0. Moreover, the resource causing the blocking can be a global resource; this

is because, under MSRP, τ_{ll} remains non-preemptive when either requesting or executing the resource. Alternatively, it might be a local resource with a ceiling priority higher than that of τ_i, enabling τ_{ll} to execute with a boosted priority, thereby blocking τ_i. This condition is denoted as $Pri(r^x, P(\tau_i)) \geq Pri(\tau_i)$, where $Pri(r^x, P(\tau_i))$ gives the priority ceiling of r^x when the resource is local to processor $P(\tau_i)$.

$$F^A(\tau_i) \triangleq \{r^x | N_{ll}^x > 0 \wedge (r^x \text{ is global} \vee Pri(r^x, P(\tau_i)) \geq Pri(\tau_i)\} \tag{18}$$

Considering that the resource can be global, multiple tasks on different processors may request r^x concurrently, thus facilitating τ_{ll} to incur *direct spin delay*. As delineated in Equation (19), α_i^x encompasses a set of processors. A processor P_a can be included in α_i^x if and only if the number of its remote requests, denoted by $Nr_a^x(R_i, R_j)$, surpasses the sum of local requests, given by $N_i^x + Nh_i^x(R_i)$. This stipulation arises because a maximum of $N_i^x + Nh_i^x(R_i)$ remote requests have been acknowledged as spin delay and incorporated into E_i as presented in Equation (12). Only processors with additional requests, such as the requests on P_2 depicted in Figure 4—which tallies five requests, with three accounted for in Equation (12) and the remaining two potentially imposing arrival blocking on τ_i—are considered. Moreover, $P(\tau_i)$ must also be accounted for in α_i^x since it contains requests that τ_{ll} sends to r^x. The term $|\alpha_i^x|$ subsequently defines the aggregate number of requests capable of inducing arrival blocking on τ_i. All necessary notations of B_i is summarized in Table 5.

$$\alpha_i^x \triangleq \{P_a | Nr_a^x(R_i, R_h) > (N_i^x + Nh_i^x(R_i)) \wedge P_a \notin P(\tau_i)\} \cup P(\tau_i) \tag{19}$$

The new RTA of MSRP meticulously delineates the resource requests emanating from each core, thereby mitigating the recurrence of spin delay calculations substantially. Despite introducing additional complexity, this approach curtails

the pessimism traditionally inherent in the analysis, fostering a notable enhancement in system schedulability [104].

Table 5: Table of Notations for Advanced MSRP (B_i)

Notation	Description
$\hat{e}_i$	Maximum arrival blocking that task τ_i can incur
$F^A(\tau_i)$	Set of resources potentially causing arrival blocking to τ_i
α_i^x	Set of processors influencing the arrival blocking on τ_i for resource r^x

2.2.5 Summary

This section reviews resource sharing in real-time systems, emphasizing the intricacies of shared resources in multicore settings. Lock-based mechanisms, especially spin-based locks, are highlighted for ensuring mutual exclusivity. The FIFO order's role in managing access sequences is discussed. The section delves into MSRP and contrasts the original MSRP RTA with the advanced MSRP RTA, underscoring advancements in WCRT calculations for a preciser system behavior analysis.

2.3 Real-Time Mixed Criticality Systems

In this section of the literature review, we explore the foundational concepts and frameworks in MCS. We start with the general definitions that govern MCS, followed by a RTA of traditional MCS model. The focus then shifts to fault-tolerance strategies, emphasizing the role they play in ensuring system robustness. Lastly, we review the recent research on fault-tolerant MCS with shared resources, highlighting both the advancements and the existing limitations in

this area. This discussion aims to provide a comprehensive overview, setting the groundwork for deeper exploration in the subsequent sections of the thesis.

2.3.1 Defintions of MCS

MCS have emerged as a pivotal concept in the domain of embedded and real-time systems. These systems are characterized by the coexistence of tasks with varying levels of criticality, all executing on a shared computational platform [20]. The term *criticality* in this context refers to the importance or severity of a task's failure to meet its deadline or functional requirements.

MCS are extensively utilized in industries such as aviation, automotive, and healthcare. These systems are responsible for handling tasks that have varying levels of safety requirements. Specifically in aviation, the DO-178C standard [102] serves as a prime example of MCS application. It categorizes tasks into five levels of criticality: Level A (Catastrophic), Level B (Hazardous), Level C (Major), Level D (Minor), and Level E (No Effect). Each level determines the extent of certification needed, based on the potential impact of a failure. This method ensures that software used in aviation is thoroughly evaluated for safety. The more critical a task is, the more rigorous the testing it undergoes. Such a structured approach is vital in systems where tasks vary significantly in importance and potential impact, ensuring that each task is managed with the appropriate level of scrutiny.

The design philosophy behind MCS revolves around a delicate equilibrium: on one hand, there is the need to optimize resource utilization, and on the other, an imperative to uphold the sanctity of task criticality. It is a balancing act where high-criticality tasks must always take precedence, ensuring they meet their stringent specifications [34].

This assurance gains heightened significance in sectors such as avionics, au-

tomotive systems, and medical devices, where any lapse in high-criticality task execution can lead to catastrophic outcomes. In certain scenarios, this might necessitate sidelining tasks of lower criticality. However, sidelining does not imply complete abandonment. Consider, for instance, an avionics system where the primary task is to maintain the aircraft's stability. In a situation where computational resources are strained, secondary tasks like in-flight entertainment might experience reduced performance or temporary unavailability.

Graceful degradation is often pursued in MCS: while the system prioritizes the most critical functions, it still aims to provide a baseline Quality of Service (QoS) for less critical tasks, albeit at a diminished capacity. Such methodical degradation in service ensures that while the system remains resilient and safety-critical operations are uncompromised, other functionalities are not entirely jettisoned but are instead scaled down in a controlled manner.

In this thesis, we consider $\mathcal{N}$ criticality levels. The criticality level of a system task τ_i is defined by the system engineer according to their importance, denoted as $l_i \in \{A, B, ..., \mathcal{N}\}$ in which A is the lowest criticality and $\mathcal{N}$ is the highest. Tasks being allocated to higher criticality levels implies a severe consequence for overall system performance if their execution in some way fails. Tasks in MCS have a series of WCET denoted as $\overrightarrow{C_{\tau_i}}$. The verification is more conservative for a higher criticality level [11], hence $C_{\tau_i,A} \leq C_{\tau_i,B} \leq ... \leq C_{\tau_i,\mathcal{N}}$. The task τ_i with criticality l_i can execute up to C_{τ_i,l_i} from its $\overrightarrow{C_{\tau_i}}$.

2.3.2 Conventional MCS Model

Baruah et al. [11] introduced the Adaptive Mixed Criticality (AMC) model, a notable MCS model in the context of preemptive fully-partitioned systems with FPS [56]. As illustrated in Figure 5, the AMC model differentiates between two system modes; the mode of the system is denoted as $\mathcal{L} \in \{LO, HI\}$. These

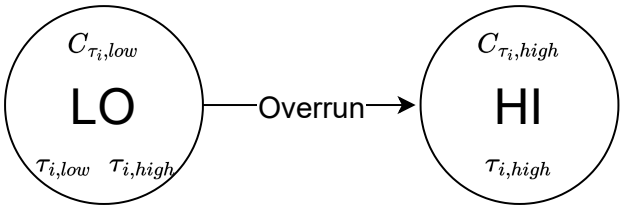


Figure 5: The AMC model

modes cater to systems with tasks that fall into one of two criticality levels. The criticality of a random task τ_i in the system is denoted as $l_i \in \{low, high\}$.

Tasks with a low criticality (i.e. $l_i = low$) run using a WCET of $C_{\tau_i,low}$. On the other hand, high-criticality tasks (i.e. $l_i = high$) have two WCETs, denoted as $\overrightarrow{C_{\tau_i}} = \{C_{\tau_i,low}, C_{\tau_i,high}\}$, where $C_{\tau_i,low} \leq C_{\tau_i,high}$. As shown in Figure 5, the system starts in the LO mode, allowing all tasks to execute up to $C_{\tau_i,low}$. If a high-criticality task surpasses its allocated budget, the system switches to the HI mode. In this mode, tasks with $l_i \geq high$ are allowed an extended budget of $C_{\tau_i,high}$, while tasks with $l_i < high$ are paused. One of the foundational assumptions of the AMC model is its ability to track the runtime of tasks. Additionally, the model is designed for flexibility, capable of accommodating an array of system modes corresponding to various criticality levels.

The RTA of the AMC model, denoted as AMC-rtb, was developed by the work in [18]. The analysis of tasks in AMC can be divided into two scenarios: one where the system remains in a stable mode, maintaining either a HI or a LO mode throughout its entire execution, and another where it experiences a transition from LO to HI mode.

The WCRT of tasks in the LO mode is presented in Equation (20). This equation comprises the WCET of τ_i in the LO mode, denoted by $C_{\tau_i,\text{low}}$, and the interference from local high-priority tasks $\tau_j \in \text{lhp}(i)$ that also execute with their LO-mode execution budgets, represented by $C_{j,\text{low}}$. Meanwhile, when a

task executes in the *HI* mode throughout its entire duration, it signifies that the task has a criticality level of l_i = high. As illustrated in Equation (21), the WCRT of τ_i, represented as $R_{i,\mathrm{HI}}$, is comprised of the *HI*-mode WCET ($C_{\tau_i,\mathrm{high}}$). It can only incur interference from local high-priority tasks with a criticality level of l_i = high, denoted by $\tau_j \in \mathrm{lhpH}(i)$, which also execute with their HI-mode execution budgets $C_{j,\mathrm{high}}$.

$$R_{i,LO} = C_{\tau_i,low} + \sum_{\tau_j \in lhp(i)} \left\lceil \frac{R_{i,LO}}{T_j} \right\rceil \cdot C_{j,low} \tag{20}$$

$$R_{i,HI} = C_{\tau_i,high} + \sum_{\tau_j \in lhpH(i)} \left\lceil \frac{R_{i,HI}}{T_j} \right\rceil \cdot C_{j,high} \tag{21}$$

If a task experiences a mode switch and continues to execute, it must therefore be a high-criticality task with l_i = high. As shown in Equation (22), the WCRT of such a task consists of its HI-mode WCET $C_{\tau_i,\mathrm{high}}$. Moreover, it will incur interference from both local high-priority tasks with low and high criticalities, denoted by $lhpL(i)$ and $lhpH(i)$, respectively. For tasks in $lhpH(i)$, they continue to execute after the mode switch, and their interference frequency is calculated as $\left\lceil \frac{R_{i,\mathrm{HI}}}{T_j} \right\rceil$. Conversely, tasks in $lhpL(i)$ will be terminated after the system switches to *HI* mode. Therefore, their interference frequency is calculated as $\left\lceil \frac{R_{i,\mathrm{LO}}}{T_k} \right\rceil$, with a duration of $R_{i,\mathrm{LO}}$, which can be deduced from Equation (20). Main notations about MCS are summarized in Table 6.

$$R_{i,switch} = C_{\tau_i,high} + \sum_{\tau_j \in lhpH(i)} \left\lceil \frac{R_{i,HI}}{T_j} \right\rceil \cdot C_{j,high} \; + \sum_{\tau_k \in lhpL(i)} \left\lceil \frac{R_{i,LO}}{T_k} \right\rceil \cdot C_{k,low} \tag{22}$$

2.3.3 Fault Tolerance Approaches

In the context of MCS, understanding faults and their tolerance mechanisms is pivotal. Faults in contemporary embedded systems are predominantly classified

Table 6: Notations for MCS

Symbol	Meaning
$\mathcal{N}$	Number of criticality levels
l_i	Criticality level of task τ_i
$\overrightarrow{C_{\tau_i}}$	Series of WCET for task τ_i
$\mathcal{L}$	Mode of the system
$C_{\tau_i,low}$	WCET of task τ_i in LO mode
$C_{\tau_i,high}$	WCET of task τ_i in HI mode
$R_{i,LO}$	WCRT of task τ_i in LO mode
$R_{i,HI}$	WCRT of task τ_i in HI mode
$R_{i,switch}$	WCRT of task τ_i during a mode switch
$lhpL(i)$	Set of local high-priority tasks with low criticality for task τ_i
$lhpH(i)$	Set of local high-priority tasks with high criticality for task τ_i

as either *permanent* or *transient* faults. Transient faults briefly disrupt system functionality, while permanent faults persist and are challenging to rectify. Certain software faults, also known as *bugs*, are a consequence of erroneous program design, categorizing them as permanent faults that cannot be rectified simply by restarting the operation [95]. In contrast, other software faults can be transient, precipitated by unexpected thread interference, and may be ameliorated through program restarts [75]. On the hardware front, transient faults can result from power supply fluctuations or electromagnetic interference, problems exacerbated by decreasing transistor size and operating voltage [51]. Permanent hardware faults, often due to hardware damage or wear, cannot be remedied until the faulty component is replaced. The focus of this thesis is specifically on transient faults, which can be resolved by reattempting the operation.

Redundancy techniques are widely adopted in literature to tolerate faults,

with the three key strategies being *re-execution* [1], *checkpointing* [26], and *replication* [83]. The *re-execution* approach safeguards task status at the beginning and detects faults at the end. Upon fault detection, the system invokes the roll-back technique, and the task is re-executed in its entirety. Conversely, the *checkpointing* technique introduces additional checkpoints within a task, typically dividing task execution into uniform segments. Each segment is individually scrutinized for faults, and upon fault detection, the system reverts to the most recent checkpoint, re-executing only the faulty segment. The *replication* technique creates several replicas of each task. The task and its replicas are released concurrently and execute in parallel. Once one execution completes without faults, the remaining tasks are discarded.

Conventional fault detection mechanisms focus on analyzing execution outputs. For instance, in lockstep dual-core architecture [89] or Triple Modular Redundancy architecture [8], multiple identical cores execute the same code, and the system applies a majority vote to pinpoint the faulty component. Acceptance tests are frequently applied at checkpoints to determine operation correctness by examining conditions that are expected to be met if the program has executed correctly [80]. In contrast, another type of fault-detection mechanism concentrates on detecting the stimuli of the fault rather than the computation results. For example, acoustic wave detectors are employed in the hardware architecture [93] to detect particle strikes that can result in transient faults during computation. Rather than using built-in hardware for fault detection, the Argus approach [73] utilizes detection equipment to monitor circuit variations. Detailed descriptions and comparisons of such types of detection mechanisms can be found in [94]. We consider transient faults which can be resolved by redundancy approaches (*e.g.* re-execution and replication) in this thesis. Each fault can only affect one task at a time and the acceptance test is applied as the fault-detection technique.

2.3.4 Fault-Tolerance and Shared Resources in MCS

Beyond addressing the fundamental challenges of MCS, researchers have explored intertwined complexities such as fault-tolerance and energy consumption. Pathan [78] introduces the FTMC, a mixed-criticality fault-tolerant algorithm for systems with two criticality levels. In FTMC, a system transition from a LO mode to a HI mode is triggered when either an overrun occurs or when the number of transient faults exceeds a predefined threshold. Chen et al. [26] propose the FTS-RHS, an online fault-tolerant MCS scheduling framework that applies checkpointing recovery schemes, thereby enhancing performance in scheduling. Building upon these efforts, Safari et al. [83] incorporate energy consumption considerations into their research on fault-tolerant MCS, proposing a LETR-MC scheme for systems with two criticality levels.

With respect to shared resources, Burns [17] applies the OPCP to MCS on a uni-processor platform with two criticality levels. Low-criticality resource holders operating at ceiling priority are suspended when the system upgrades to the HI mode. These suspended tasks can resume execution by inheriting the execution budget from the next task trying to access the resource. Zhao et al. [103] extend the Priority Ceiling Protocol (PCP) [87] to the Highest-Locker Criticality, Priority-Ceiling Protocol (HLC-PCP) to manage resource sharing in the MCS under the AMC scheme. Han et al. [47] then migrate the MSRP to the MCS and develop a criticality-aware utilization bound.

More recently, research into fault tolerance has been extended to support MCS. Al-bayati et al. [1] propose a state-of-the-art fault-tolerant model to tackle both task overruns and transient faults for MCS with tasks' criticality levels $l_i \in \{low, high\}$, known as the *four-mode* model. Based on the AMC model, the four-mode model introduces two additional system execution modes: TF (transient faults) and OV (overruns) to deal with transient faults and task overruns

separately.

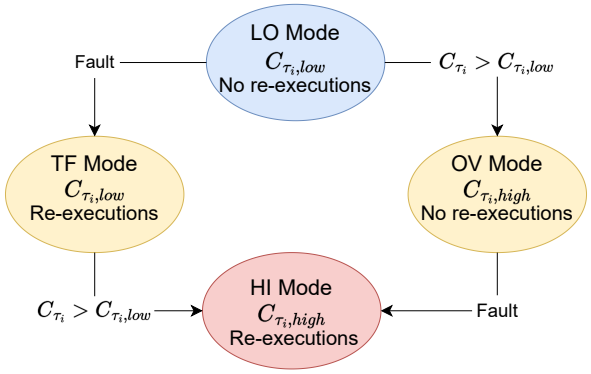


Figure 6: The four-mode model [1]

Figure 6 illustrates the system execution flow of the four-mode model. The system initially starts in LO mode, in which each task is running with a budget $C_{\tau_i,low}$. If any high-criticality task incurs a transient fault, the system moves to TF mode, the *re-execution* approach is then applied to resolve the fault. Or, if any task overruns its budget in the LO mode the system will move to the OV mode, where high-criticality tasks can execute with a higher budget $C_{\tau_i,high}$. Lastly, the system moves to the HI mode from TF or OV, if an overrun or a fault occurs, respectively. In HI mode, tasks can execute up to $C_{\tau_i,high}$ with re-executions allowed for resolving faults.

The four-mode model provides reliability and timing guarantees for dual-criticality MCS without any shared resource. However, the four-mode model has the following three major limitations.

Limitation 1. *The four-mode model does not scale well for systems with more than two criticality levels.*

With the four-mode model, a task overrun can bring the system from LO to OV mode whereas a task fault advances the system from LO to TF mode. Then, at OV and TF modes, the model requires the occurrence of both task fault

and overrun to advance the system to the HI mode. Such design is sufficient for systems with two criticality levels conceptually. However, for systems with multiple criticality levels (*e.g.* four criticality levels in ISO 26262 [59] and five criticality levels in the avionic standard DO178-C [101]), the four-mode model becomes difficult (or in reality impossible) to apply as the model requires the occurrence of both transient faults and task overruns to advance to the next mode. For example, in a system with five levels of criticality, the AMC model can adopt five system modes (*e.g.* $A - E$) and four occurrences of task overrun will take the system to E mode, whereas the four-mode model only reacts to the first overrun and neglects the rest, due to the absence of transient faults.

Limitation 2. *The four-mode model relies on re-executing the entire task to resolve faults. With the presence of shared resources, this can lead to prolonged blocking for the faulty task, increase the interference of all low priority tasks, and result in low system schedulability.*

In particular, if a fault occurs during execution without the involvement of shared resources, re-executing the whole task is very pessimistic due to the additional blocking imposed to both the task itself and other tasks in the system. Moreover, re-execution indicates re-requesting every resource access, the task can block (or be blocked by) other tasks that are requesting the same shared resource again [109].

Limitation 3. *The four-mode model relies on the re-execution approach, which cannot immediately resolve faults occurring in a critical section. This can cause transitive effects if the correctness of other tasks depends on the computation results of the resource.*

When a fault occurs during the critical section of a task with re-execution applied, it cannot be resolved immediately because the detection is applied at

the end of a task. Therefore, the erroneous computation results submitted by a task to a shared resource could be used by the following tasks that share the same resource, which can incur undesired domino effects and impose a threat to the reliability of the whole system. Hence, a fault-tolerant approach that can address faults in shared resources immediately before the lock is released should be adopted.

2.3.5 Summary

In MCS literature, tasks with different criticalities are addressed, especially in sectors like avionics where precise task execution is crucial. Baruah et al. proposed the AMC model, categorizing tasks into low or high criticalities and adjusting their performance accordingly. Although fault tolerance is essential in MCS, the four-mode model by Al-bayati et al. revealed some shortcomings. It struggles with systems having multiple criticality levels and is not always efficient in managing faults. This highlights the need for more advanced fault-tolerance models in MCS that can effectively manage shared resources. These challenges pave the way for the proposed HPRTS framework, as outlined in Chapter 3, aiming to offer a fault-tolerance solution for multiprocessor MCS with shared resources.

2.4 Task Allocation Methods

Task allocation is a crucial part of real-time systems, ensuring that tasks are efficiently distributed across processors to meet specific timing and performance requirements. Over the years, many methods have been developed to tackle challenges posed by the increasing complexity of systems and changing workloads. This section delves into the history and evolution of task allocation methods, moving from basic static approaches to advanced dynamic strategies. We then

shift our focus to allocations designed for systems with shared resources. Here, we will discuss the advantages, limitations, and details of different contention-aware allocation techniques.

2.4.1 Evolution of Task Allocation Methods

Task allocation in real-time systems is an essential aspect that has garnered significant research attention. In the early stages, real-time systems primarily employed static task allocation [68]. In this scenario, tasks were assigned to specific processors during the design phase and remained fixed throughout the system operation. While straightforward and predictable, this method best suited real-time systems with a stable, unvarying workload.

However, as systems grew more complex and workloads became less predictable, the limitations of static task allocation began to surface. To address these constraints, the concept of dynamic task allocation was introduced, leading to the evolution of fully partitioned scheduling systems [30].

In fully-partitioned systems, tasks are statically assigned to designated cores and cannot migrate during execution [70]. This method promotes predictable scheduling behavior, minimizes run-time overhead, and simplifies inter-task synchronization management. However, it may lead to sub-optimal system utilization when dealing with variable or unpredictable workloads.

To optimize fully partitioned scheduling, various methods have been proposed. Heuristic methods such as Best-Fit (BF), First-Fit (FF), Worst-Fit (WF) and Next-Fit (NF) are among these [3]. The BF strategy allocates tasks to the processor where the task's requirement best fits the remaining capacity, minimizing resource wastage. The FF strategy assigns tasks to the first processor with sufficient capacity, thus expediting the allocation process. The WF strategy places tasks on the processor with the most remaining capacity, aiming to

balance load across processors. NF sequentially assigns tasks to processors. If a processor lacks capacity for a task, the allocator proceeds to the next processor. The process continues until an appropriate processor is identified or all are examined. The last assigned processor is noted for subsequent allocations. Heuristic methods are favored for their simplicity and computational efficiency. However, their performance can be limited as system complexity increases especially with the inclusion of shared resource [106].

Apart from heuristic methods, mathematical programming techniques such as Integer Linear Programming (ILP) can be employed for task allocation [44]. ILP aims to provide an optimal solution, albeit with high computational complexity, and is thus suited for systems with a smaller number of tasks. Metaheuristic methods like Genetic Algorithms also find use in more complex scenarios, offering near-optimal solutions with lower computational demands at the cost of potential imprecision [44].

2.4.2 Resource-Aware Task Allocation Methods

In the study of multiprocessor architectures, resource-aware task allocation is key. It focuses on managing tasks effectively, taking into account the need for mutually exclusive access to shared resources [27, 48, 49, 55, 57, 62, 97, 99, 100]. This need influences the strategies and approaches in task allocation, forming the basis of different scheduling algorithms.

Building on the basic concepts discussed earlier, we now focus on resource-aware allocation methods designed for fully-partitioned FPS systems. These methods aim to work well in settings where exclusive access to shared resources is a must, helping to reduce the blocking time during resource access and, as a result, improving system efficiency.

Executing Critical Section on Host Core

Currently, most methods execute critical sections on the host processors of tasks [48,49,55,62,97]. In this context, Lakshmanan et al. [62,104] introduced the Synchronization-aware Partitioning Algorithm (SPA). This algorithm groups all tasks that either directly or indirectly share the same resources and assigns these groups of tasks to processors using a strategy known as the Best-Fit Decreasing (BFD) heuristic. This strategy allocates tasks using the best-fit approach, following a non-increasing utilization order of tasks. The idea behind SPA is to allocate tasks that share the same resource in the same bundle and to allocate that bundle to a single processor, thereby localizing the shared resources and reducing the blocking incurred due to accessing global resources. A detailed implementation of SPA is explained below:

Step 1: Initialization

- Bundle Creation: Form bundles for tasks sharing the same or transitive resources.
- Handle independent tasks separately.

Step 2: Initial Allocation

- Task and Bundle Sorting: Order tasks/bundles by decreasing utilization.
- Utilization of BFD Algorithm: Use BFD algorithm for allocation with a utilization cap per core.
- Prioritize allocatable tasks/bundles, defer others.

Step 3: Breaking and Reallocating Macrotasks

- Breaking Cost Calculation: Calculate the cost to break each unallocatable bundle.

- Bundle Sorting: Sort bundles by increasing breaking cost.
- Bundle Breaking and Creation: Break least costly bundle and form a new one matching available utilization.
- Allocation and Core Addition: Allocate broken bundles, adding cores as needed.

The SPA is a method characterized by its versatility, fundamentally operating based on the knowledge of resource usage. However, it encounters a significant limitation in its approach to task bundling and allocation. In SPA, tasks are grouped into the same bundle simply if they share the same resources, without considering other vital details. This approach can lead to the creation of oversized bundles, particularly in systems with extensive resource sharing. When a bundle becomes too large to be assigned to a single core, it necessitates breaking down the bundle, a process that primarily considers the system's utilization, often overlooking the actual levels of resource contention between tasks. This can result in tasks, which have higher levels of resource contention, being separated, thereby reducing the efficiency in contention reduction. The cirtical issue is summarized below.

Limitation 4. *The SPA [62] can create large bundles that no processor can accommodate, leading to a breakdown process that does not always effectively reduce resource contention, potentially resulting in inefficiencies [57, 97].*

Migrating to Access Shared Resoruce

Rather than executing critical sections on the host processor, Hsiu et al. [55] suggest a method that allocates resources to processors and migrates a task to a specified processor each time it requests a shared resource. This strategy also

localizes shared resources and can reduce resource contention between processors. As reported, the method [55] reduces the number of processors needed for system scheduling. In the same vein, a Resource-Oriented Partitioning (ROP) method is developed in [57, 100]. The ROP method allocates tasks and shared resources to different sets of processors, namely synchronization and application cores. In this setup, synchronization cores are dedicated to managing access to shared resources, thereby localizing the contention and reducing the associated overhead. Meanwhile, application cores are primarily tasked with executing the application tasks. This segregation facilitates a reduction in resource contention and optimizes the scheduling process by allowing tasks to migrate to synchronization cores when accessing shared resources, and thereby maintaining a high level of control over resource access protocols. The method leverages specialized schedulability tests and offers two variations that employ either the priority ceiling protocol or the non-preemptive protocol to govern resource access, enhancing the system's overall efficiency and schedulability. The allocation process of ROP is delineated in the subsequent steps.

Step 1: Initialization

- Analyze task resource utilization, including access count and duration.
- Sort resources by utilization and tasks by deadlines.

Step 2: Priority Assignment

- Prioritize tasks based on deadline order, with shorter deadlines higher.

Step 3: Resource Allocation

- Adjust synchronization processors from one to minimum of tasks or partitions to find a feasible solution.
- Allocate resources on processors using WFD, balancing load and total utilization.

Step 4: Task Allocation

- Create and track an empty allocation list for each processor.
- – Sequentially select tasks for allocation.
  - – Try allocating to non-synchronization processors first.
  - – Ensure task can be scheduled on chosen processor with RTA.
  - – Allocate and update task details if schedulable.

Step 5: Finalization

- Verify all tasks are allocated.
- – Return scheme if successful.
  - – Return null if not.

Step 6: Iterative Adjustments

- Modify parameters and retry until a solution is found or all options are exhausted.

The ROP method, grounded in a search-based approach, distinctly allocates tasks and shared resources to synchronization and application cores, fostering a more efficient real-time system scheduling. This strategy not only localizes the resource contention, reducing the associated overhead but also leverages specialized schedulability tests to find an optimal allocation of tasks and resources, enhancing the system's performance. By allowing tasks to migrate to synchronization cores for accessing shared resources, it minimizes conflicts and promotes a smoother execution flow, potentially reducing the number of processors required for system scheduling. The search-based nature of ROP ensures a detailed exploration of possible solutions, aiming to find the most optimized

allocation, albeit at the cost of higher computational time. It is demonstrated in [100] that ROP outperforms a set of existing methods, including greedy slacker approach in [97].

However, ROP is a search-based allocation algorithm, it maps tasks to processors by iterating through each one and assessing the system's schedulability using a specific test. Although search-based allocation often produces better schedulability results compared to conventional heuristic approaches (e.g., WF), potentially enhancing system performance, it can also prolong implementation time. This leads to the following limitation.

Limitation 5. *Search-based task allocation methods that rely on schedulability tests, such as ROP [57], can result in extended computation times, making them unsuitable for runtime applications.*

Tasks under ROP may incur additional migration overhead each time they access a shared resource—i.e., migrating to and returning from a dedicated processor. This can introduce a significant runtime overhead, leading to the subsequent limitation [104, 109].

Limitation 6. *In scenarios with intensive resource access, ROP [57] can induce frequent task migrations, imposing considerable overhead and thereby reducing the method's performance [109].*

2.4.3 Summary

Task allocation in real-time systems has transitioned from static to dynamic strategies to optimize the distribution of tasks across processors. Allocation in multiprocessor environments with shared resources has attracted many attentions. While methods such as SPA group tasks by shared resources, they can occasionally produce oversized bundles, resulting in inefficiencies. Conversely, the

ROP method, which assigns tasks and resources to distinct processors, can result in longer computation times because of its iterative approach. Additionally, it introduces overhead from frequent task migrations. These challenges underscore the necessity for enhanced task allocation techniques and form the basis for the second step of the proposed HPRTS framework as presented in Chapter 4.

2.5 DAG Tasks in Real-Time Systems

In real-time systems, the intricacy of tasks and their dependencies has increased significantly. DAG are now essential tools for representing and managing these relationships, facilitating effective task execution on multicore platforms. This section provides an overview of DAG tasks, from basic models to advanced scheduling methods and RTA. As we explore DAG tasks further, we identify the challenges that require solutions.

2.5.1 Generic DAG Task Model

As real-time systems continue to grow in both scale and complexity, the tasks within these systems have begun to exhibit intricate internal structures, necessitating solutions that can leverage high performance to manage these complexities adeptly. A representation that has gained prominence in this context is that of DAG, wherein nodes represent subtasks, and edges signify the precedence constraints between them. This representation is increasingly utilized in real-time systems due to its ability to accurately depict complex task dependencies, thereby facilitating high-performance parallelism on multicore platforms.

We assume a multi-DAG system contains z recurrent DAG tasks $\Gamma = \{\tau_1, ..., \tau_z\}$ as illustrated in Figure 7, DAG tasks in this thesis are independent of each other, that is we assume nodes of two different DAG tasks do not share any resources and there exists no dependency between them. According to [53,108], a real-time

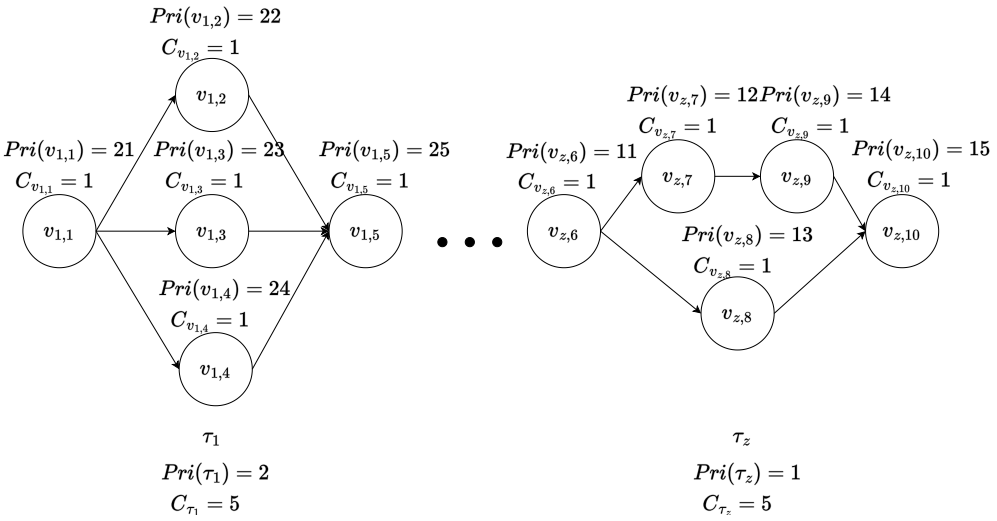


Figure 7: An example of multiple DAG tasks.

DAG task τ_i is defined by following tuples $\{T_i, D_i, \mathcal{G}_i = (V_i, E_i), Pri(\tau_i)\}$. Here, T_i, D_i, and $Pri(\tau_i)$ are as defined in the Section 2.1.1. Moreover, $\mathcal{G}_i$ represents a graph that defines the set of activities forming the DAG task. This graph is described as $\mathcal{G}_i = (V_i, E_i)$, where V_i is the set of nodes and $E_i \subseteq (V_i \times V_i)$ indicates the set of directed edges connecting any two nodes.

A node in DAG τ_i is represented as $v_{i,j} \in V_i$, with the index j specifying the node and the index i indicating its affiliation to τ_i (e.g., $V_1 = \{v_{1,1}, v_{1,2}, v_{1,3}, v_{1,4}, v_{1,5}\}$ as presented in Figure 7). For simplicity, the subscript of the DAG task (i.e., i for τ_i) is dropped when a single DAG task is under consideration. Each node v_j are assigned a unique priority, represented as $Pri(v_j)$, respectively. Furthermore, if the priority of τ_i exceeds that of τ_j (i.e. $Pri(\tau_i) > Pri(\tau_j)$), then every node in τ_i will have a priority greater than those in τ_j (i.e. $Pri(v_a) > Pri(v_b)$, $\forall v_a \in V_i$ and $v_b \in V_j$). The WCET of a node v_j is given as C_{v_j}, while the cumulative WCET of a DAG τ_i is represented as $C_{\tau_i} = \sum_{v_j \in V_i} C_{v_j}$.

If two nodes v_j and v_k are connected by a directed edge (i.e., $(v_j, v_k) \in E$), v_k can start executing only if v_j finishes executing. That is, v_j is a *predecessor*

of v_k, whereas v_k is a *successor* of v_j. The predecessors and successors of node v_j can be formally defined as $pre(v_j) = \{v_k \mid v_k \in V_i \wedge (v_k, v_j) \in E\}$ and $suc(v_j) = \{v_k \mid v_k \in V_i \wedge (v_j, v_k) \in E\}$, respectively. Nodes that are either *direct* or *transitive* predecessors and successors of a node v_j are termed as its ancestors $anc(v_j)$ and descendants $des(v_j)$ respectively. A node v_j with $pre(v_j) = \varnothing$ or $suc(v_j) = \varnothing$ is referred to as the *source* v_{src} or *sink* v_{sink} respectively. As with most of the existing work [53, 54, 107], we assume a generic DAG model that always starts with one source and ends with one sink node, otherwise, a dummy node without utilization is added to the beginning or the end of the DAG.

Based on the DAG task model. we define the features of a DAG that will be utilized by RTA constructed in this thesis. Table 7 summarizes main notations of DAG tasks.

Definition 1. *When bounding RTA for DAG tasks, the WCRT of a DAG in a single-DAG system is often referred to as the* ***makespan***

Definition 2. *(Concurrent nodes) For a pair of nodes $v_j, v_k \in V_i$, if v_j is neither the ancestors of v_k (i.e. $v_j \notin anc(v_k)$) nor the descendants of v_k (i.e. $v_j \notin des(v_k)$), it is referred to as a concurrent node of v_k. The set of concurrent nodes of v_j is denoted as $S(v_j)$ and is expressed as Equation 23.*

$$S(v_j) = \{v_k | v_k \in V_i \wedge v_k \notin anc(v_j) \wedge v_k \notin des(v_j) \wedge v_k \neq v_j\} \tag{23}$$

Example 5. *Taking the DAG task shown in Figure 8 as an example. The concurrent nodes of v_2 within the DAG is $S(v_2) = \{v_3, v_4, v_5\}$.*

A complete path within a DAG is defined as follows. As no partial paths will be used in this thesis, we use the term path directly.

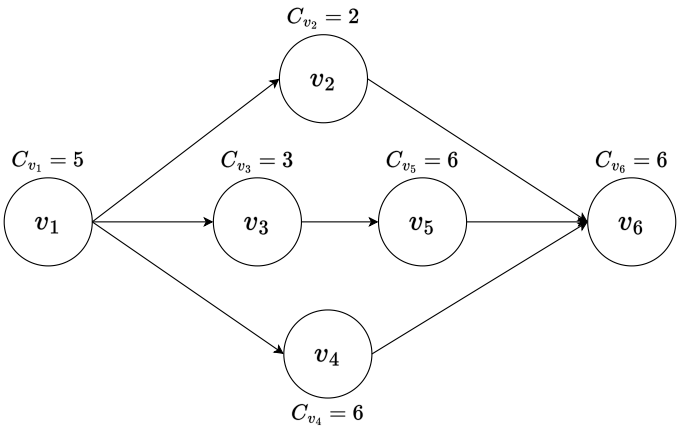


Figure 8: An example of a DAG task τ_i

Definition 3. *(Path) A path of τ_i is denoted by $\lambda_a = \{v_0, v_1, \cdots, v_k\}$, where $\forall p \in [0, k], (v_p, v_{p+1}) \in E$, $v_0 = v_{src}$ and $v_k = v_{sink}$.*

For a path λ_a, C_{λ_a} returns its total workload (i.e., length), which is the sum of the WCET of nodes in λ_a. The path with the highest workload (C_{λ_a}) within a DAG τ_i is defined as the critical path and is denoted as λ_i^*.

Example 6. *As shown in Figure 8, the DAG task consists of three paths: $\lambda_1 = \{v_1, v_2, v_6\}$, $\lambda_2 = \{v_1, v_3, v_5, v_6\}$ and $\lambda_3 = \{v_1, v_4, v_6\}$, with $C_{\lambda_1} = 5 + 2 + 6 = 13$, $C_{\lambda_2} = 5 + 3 + 6 + 6 = 20$ and $C_{\lambda_3} = 5 + 6 + 6 = 17$, respectively Therefore, λ_2 is the critical path of this DAG task.*

2.5.2 DAG Task Scheduling

Similar to the traditional sporadic task model, research on DAG task scheduling has branched into partitioned, global, and federated scheduling, as introduced in section 2.1.2. Among these, global scheduling has garnered the most attention, becoming a significant focus in research. Beyond offering improved system utilization, it provides more flexible and adaptable scheduling solutions adept at

Table 7: List of Notations for DAG Task

Notation	**Definition**
$\mathcal{G}_i$	Graph representing set of activities in τ_i
V_i	Set of nodes in τ_i
E_i	Set of directed edges in τ_i
$v_{i,j}$	Node j in DAG τ_i
$Pri(v_j)$	Priority of node v_j
C_{v_j}	WCET of node v_j
λ_a	A path in τ_i
C_{λ_a}	Total workload of path λ_a
λ_i^*	Critical path in τ_i
$pre(v_j)$	Predecessors of node v_j
$suc(v_j)$	Successors of node v_j
$anc(v_j)$	Ancestors of node v_j
$des(v_j)$	Descendants of node v_j
v_{src}	Source node of a DAG
v_{sink}	Sink node of a DAG
$S(v_j)$	Set of concurrent nodes of v_j

managing dynamic changes. This strategy is especially advantageous in environments where tasks have precedence constraints modeled as a DAG [9].

Global list scheduling entails maintaining a global list of all tasks [72]. A pivotal step in global list scheduling is task prioritization. Tasks receive priorities from metrics like computational demands and task dependencies. This prioritization can either be static, set before execution, or dynamic, allowing for adjustments during runtime. The aim is to minimize the overall execution time, or the makespan, through the smart allocation of tasks to processors [79].

At the core of this method is the upkeep of a global list of all tasks awaiting scheduling. This list acts as a repository, guiding the selection of tasks for scheduling and ensuring a consistent view of all tasks. In a DAG setting, the node with the highest priority might not be the initial choice for scheduling due to the need to honor task dependencies [58].

Work-Conserving and Non-Work-Conserving

Work-conserving scheduling stands as a strategy where the scheduler ensures that the processor is never idle if there are tasks ready to be executed [39]. In the context of global list scheduling, this means that tasks are continuously pulled from the global list and allocated to processors as long as there are tasks available. This strategy aims to maximize resource utilization and minimize task waiting times, thereby potentially increasing the overall system throughput. However, it requires a well-structured global list that can quickly identify and allocate the next task to be executed, ensuring a smooth and efficient scheduling process.

On the other hand, non-work-conserving scheduling allows for periods where the processor remains idle despite having tasks ready in the global list for execution [82]. This strategy is employed to ensure that higher-priority tasks receive the focus they require, potentially waiting for a more opportune time for execution, even if it means a temporary underutilization of resources. In a global list scheduling context, this could mean a more strategic allocation of tasks, where the scheduler can withhold lower-priority tasks in the global list to prioritize the execution of higher-priority tasks, maintaining a quality of service and meeting critical deadlines.

Preemptive, Non-Preemtpive and Limited Preemption

In the context of global list scheduling applied to DAG tasks, understanding the nuances of different scheduling strategies—namely preemptive, non-preemptive, and limited preemption scheduling—is pivotal [12]. These strategies dictate how tasks are managed and executed, each with its unique approach to handling task priorities and preemptions.

Starting with preemptive scheduling, this strategy allows for a DAG task to be interrupted at any point, provided a higher-priority task or node becomes available. This ensures that high-priority tasks are not left waiting, thereby optimizing the system's responsiveness and throughput. While this strategy offers flexibility, it does come with its set of challenges, including the potential for frequent context switches and the associated overheads.

Transitioning to non-preemptive scheduling, here, once a DAG task commences, it sees through to completion of the last node of the DAG, regardless of the emergence of higher-priority tasks. This strategy stands in contrast to preemptive scheduling, offering a simplified scheduling process at the expense of potentially longer waiting times for high-priority tasks, as it strictly adheres to the execution pattern defined by the DAG.

Bridging the gap between preemptive and non-preemptive scheduling is the strategy of limited preemption scheduling. This approach permits preemptions at the task level while retaining a non-preemptive stance at the individual node level within each task. Essentially, it seeks to marry the responsiveness of preemptive scheduling with the simplicity of non-preemptive scheduling, creating a controlled environment that strategically designates preemption points to curb the potential overhead induced by unrestricted preemptions.

Timing Anomaly in DAG

A timing anomaly in the context of DAG scheduling refers to a situation where a local improvement in the execution time of a node not lead to an improvement in the overall execution time of a DAG or a system, and can sometimes even lead to a degradation in the global performance.

In DAG scheduling, the execution of a node should obey the precedence constraint. A timing anomaly can occur in such a system due to various factors including, but not limited to, the complex interplay of task dependencies, resource contentions, and scheduling policies. Example 7 presents a timing anomaly scenario under global list scheduling with a limited preemption scheduling scheme in a work-conserving manner.

Example 7. *Figure 9 presents a DAG example and its two abstracted execution Gantt charts (Simulation and Reality). Node priorities are not necessary in this example and hence are omitted. As shown in the Simulation chart,* v_1 *starts at* $t = 0$ *on partition* P_2 *as it is the source node. When* v_1 *finishes at* $t = 1$, v_2 *and* v_3 *become eligible to execute and begin execution on* P_1 *and* P_2*, respectively. When* v_3 *finishes at* $t = 3$, v_6 *starts and continues until* $t = 8$ *on partition* P_1*. Following the completion of* v_2 *at* $t = 4$, v_4 *and* v_5 *execute sequentially on* P_2*. Finally,* v_7 *starts at* $t = 8$ *and finishes at* $t = 9$*, resulting in a simulation makespan for the DAG of 9 units. However, in reality, a node is likely to execute in less time than its WCET. In the Reality scenario,* v_2 *executes for only one unit of time, from* $t = 1$ *to* $t = 2$*, which initially appears to be an improvement. However, due to limited preemption, this alters the execution order of* v_4, v_5, *and* v_6*, resulting in a makespan of 10 units.*

To avoid timing anomalies in limited preemption environments while maintaining timing predictability, numerous studies have introduced additional runtime approaches to eliminate timing anomalies in DAG scheduling [28, 66].

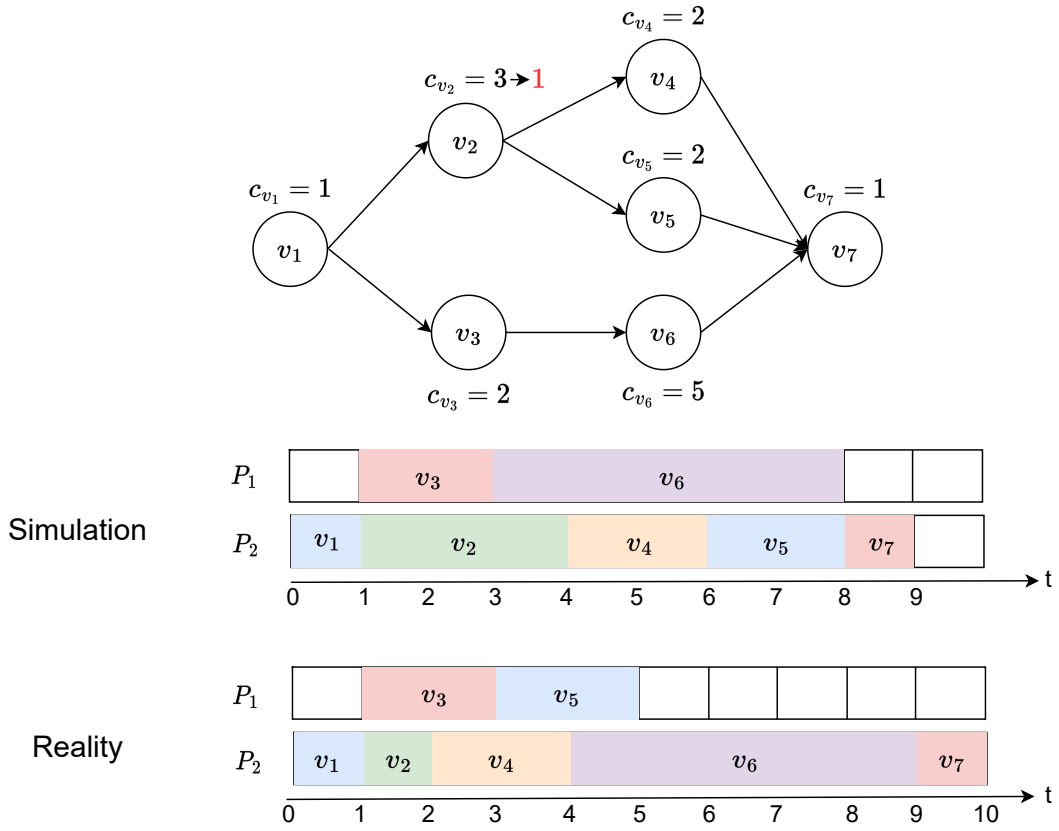


Figure 9: An example of timing anomaly

In one such approach detailed in a study [28], the authors propose a system that monitors the actual runtime execution of each execution unit within a DAG. This system strictly idles the core to preserve the pre-established execution order of DAG tasks, effectively removing timing anomalies encountered during scheduling with limited preemption. To provide timing guarantees to the system, they simulate a hyperperiod of the system and calculate the worst-case system behavior through simulation.

However, this approach has its drawbacks. Firstly, it renders the system non-work-conserving, as there are instances where ready nodes cannot execute because they must adhere to the predefined execution order, even when a core is

idle. Secondly, online monitoring introduces additional overhead to the system, potentially exacerbating system performance. On the other hand, developing a RTA to bound the WCRT of tasks can also foster timing predictability without managing DAG execution at runtime. This strategy aims to ensure that the system can reliably predict the maximum time it takes for a task to be completed, enhancing the overall predictability and reliability of the system.

2.5.3 RTA for DAG Tasks

There exists a large body of work on analyzing DAG tasks, which covers a wide range of hardware settings, scheduling schemes and task models [24,36,52,53,60, 65,74,86,107,108]. In this subsection, we focus on the major existing analysis for DAG tasks with a global work-conserving schedule in homogeneous multi-core systems. An in-depth review is provided to illustrate the major limitations and defects of the existing analysis.

Generic DAG Analysis under Work-Conserving Schedules

The majority of existing methods on DAG scheduling assume a work-conserving scheduling scheme [74]. For this schedule, Graham [45] proposes the classic generic bound for the traditional tasks, in which the interference of a task is computed by the average value of the total workload of other tasks on all processors.

Based on the classic bound, Melani et al. [74] construct an analysis to compute the WCRT of DAG tasks under a preemptive scheduling scheme. This analysis takes the priorities of the DAG tasks as an input but assumes no knowledge of the intra-node priority of each DAG. We note that the analysis is constructed for conditional DAGs, i.e., a DAG task can contain conditional edges that may not be active during execution, but is fully compatible with the unconditional ones.

Under this analysis, two key factors of the WCRT of a DAG task are identified: the intra-task and inter-task interference that can be summarized below.

Definition 4. *(Intra-task interference) the amount of interference incurred by a node from high-priority nodes within the same DAG.*

Definition 5. *(Inter-task interference) the amount of interference incurred by a node from high-priority nodes from other DAGs.*

Later on, the generic analysis is improved by Fonseca et al. [36] using a finer-grained approach for computing the inter-task interference caused by carry-in and carry-out DAGs. Following a similar approach, Serrano et al. [86] construct a generic analysis for DAG tasks for the limited preemption scheme, where a blocking factor imposed by the low-priority nodes and DAGs is introduced in the analysis. The definitions of blocking are defined below. When referring to the term *delay* by itself, it contains both interference and blocking.

Definition 6. *(Intra-task blocking) the amount of blocking incurred by a node from low-priority nodes within the same DAG.*

Definition 7. *(Inter-task blocking) the amount of interference incurred by a node from low-priority nodes from other DAGs.*

As shown in Equation 24 [86], the WCRT of a DAG can be divided into two parts: 1) the worst-case makespan of a DAG, 2) the inter-task delay suffered from other DAG tasks.

$$R_i = \underbrace{\text{Makespan}}_{\text{part 1}} + \underbrace{\text{InterDelay}}_{\text{part 2}} \tag{24}$$

As shown in Equation 25, the first part calculates the worst-case makespan of a DAG task τ_i. The $C_{\lambda_i^*}$ term is the sum of the WCET of the nodes in the critical path. The analysis includes the principle that all nodes apart from the

critical path can provide delay to the critical path. Therefore, the amount of delay incurred by the critical path is denoted as $\frac{1}{m}(C_{V_i} - C_{\lambda_i^*})$, in which C_{V_i} gives the total workload of τ_i and $C_{V_i} - C_{\lambda_i^*}$ represents the amount of workload of a DAG apart from the critical path. The workload after subtraction is then divided by the number of cores in the system m to work out the upper bound of the delay incurred from nodes within the same DAG task.

Then, the second part calculates the inter-task delay of τ_i, in which I_i^{hi} is the workload of the interference delay imposed by high-priority tasks, which is the sum of the WCETs of all high-priority nodes that are released during the execution of τ_i. As for B_i^{lo}, it contains the workload of the blocking delay from low-priority tasks due to limited preemption scheduling. To bound B_i^{lo}, Lemma 1 and 2 are constructed in [92].

$$R_i = \underbrace{C_{\lambda_i^*} + \frac{C_{V_i} - C_{\lambda_i^*}}{m}}_{\textbf{part 1}} + \underbrace{\left\lfloor \frac{I_i^{hi} + B_i^{lo}}{m} \right\rfloor}_{\textbf{part 2}} \tag{25}$$

Lemma 1. *For a DAG task, the first node can suffer a blocking delay from at most m (total core number) nodes with lower priorities [92].*

Lemma 2. *For a DAG task, nodes apart from the first node can suffer the blocking delay from at most $m-1$ nodes with lower priorities [92].*

According to Lemma 1 and 2, the classical bound then assumes the first node (resp. every node apart from the first node) can suffer blocking delay from m (resp. $m-1$) largest low-priority nodes that can execute in parallel [86]. The largest m (and $m-1$) nodes among the DAG tasks with lower priorities that can execute in parallel are denoted as δ_i^m (and δ_i^{m-1}). As shown in Equation 26, B_i^{lo} is equal to the total workload of δ_i^m (incurred by the first node) plus $|V_i|-1$ times the workload of δ_i^{m-1} (incurred by rest of the nodes), where $|V_i|$ represents

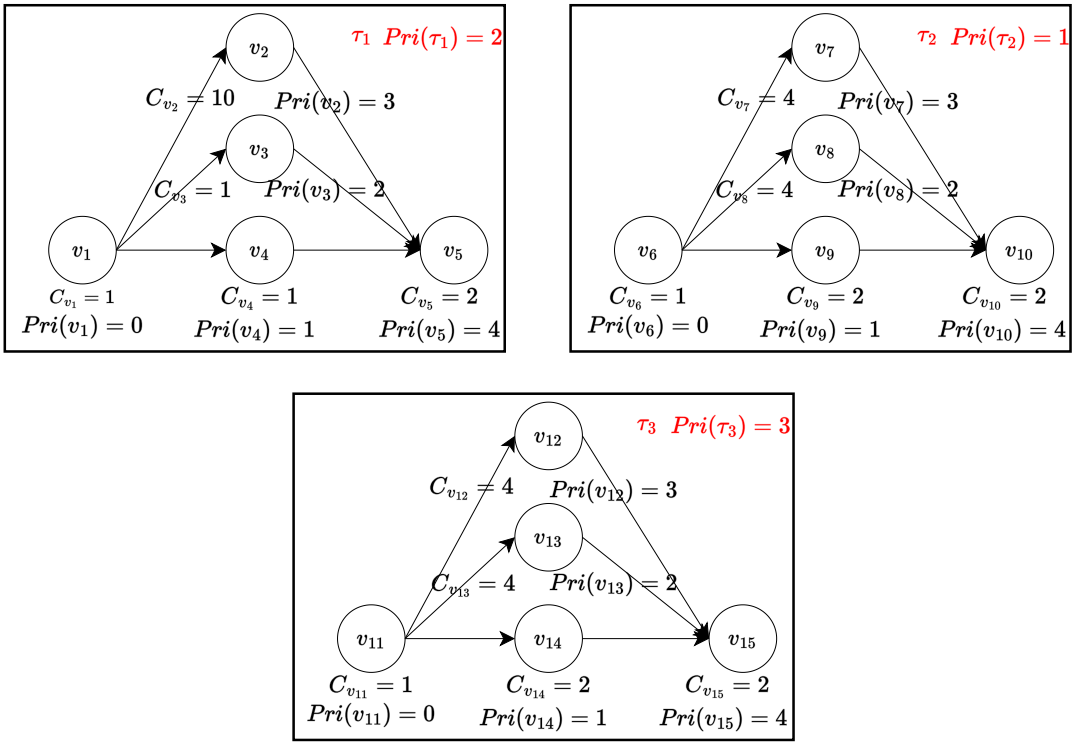


Figure 10: Three DAG tasks with different priorities.

the size of the nodes in τ_i

$$B_i^{lo} = C_{\delta_i^m} + (|V_i| - 1) * C_{\delta_i^{m-1}}; \tag{26}$$

An example of the calculation of the classical bound [86] is presented below.

Example 8. *We first calculate the worst-case makespan of* τ_1 *shown in Figure 10, which is the first part of Equation 25. The critical path of* τ_1 *is* $\lambda_1^* = \{v_1, v_2, v_5\}$. *The total workload of the* λ_1^* *is* $C_{\lambda_1^*} = 1 + 10 + 2 = 13$. *The total workload of* τ_1 *is* $C_{V_1} = 1 + 10 + 1 + 1 + 2 = 15$. *For a dual-core system, the interference of* λ_1^* *can be calculated as* $\frac{15-13}{m} = \frac{15-13}{2} = 1$. *The worst-case makespan is equal to* $R_1 = C_{\lambda_1^*} + 1 = 13 + 1 = 14$.

Then, we calculate the inter-task delay. For simplicity, assuming τ_2 *and* τ_3 *will only release once during the execution of* τ_1. *Task* τ_3 *has higher priority than*

τ_1, *i.e* $Pri(\tau_3) = 3 > Pri(\tau_1) = 2$, *hence* $I_1^{hi} = C_{V_3} = 1+4+4+2+2 = 13$. *Task* τ_2 *has lower priority than* τ_1, *We have* $\delta_1^2 = \{v_7, v_8\}$ *and* $\delta_1^1 = \{v_7\}$, *then* $C_{\delta_1^2} = 4+4 = 8$ *and* $C_{\delta_1^1} = 4$, *the size of* τ_1 *the* $|V_1| = 5$. *Finally,* $B_1^{lo} = 8+(5-1)*4 = 24$. *According to Equation 25, we have* $R_1 = 14 + \lfloor \frac{13+24}{2} \rfloor = 32$

Based on the description provided above, we can see that the classical bound for systems with limited preemptions proposed in [86] is rather pessimistic. The main limitations of the classical bound can be concluded as follows.

Limitation 7. *The analysis neglects the parallelism feature between nodes and results in a pessimistic calculation of the worst-case response time.*

For example, for the analysis of a single DAG, the classical bound assumes all the nodes apart from the critical path will delay with the critical path. However, some nodes can execute in parallel without delaying the execution of the critical path. Moreover, when accounting for the inter-task delay, it also assumes all nodes with a higher priority will interfere with the DAG. For a system with sufficient cores available, DAG tasks can also execute in parallel without interfering with each other.

Limitation 8. *When calculating the inter-task low-priority blocking, each node apart from the source node is assumed to incur a constant low-priority blocking from* δ_i^{m-1}, *which is pessimistic.*

According to Lemma 2, each node apart from the source node will suffer blocking from at most $m-1$ low-priority nodes. However, each low-priority node can only delay a high-priority node once during one release [21]. Thus, assuming nodes in δ_i^{m-1} can constantly block all nodes apart from the first node will result in repetitive calculations of blocking.

Moreover, the Limitation 8 is also caused by that fact that intra-task and inter-task blocking are analyzed separately, which can inevitably cause redundant calculations.

DAG Analysis with Explicit Intra-task Priority

Despite its utility, this generic bound can become overly pessimistic, especially when the precise node execution order is predetermined before system execution, leading to an overestimation of interference and blocking [108].

He et al. [54] alleviates pessimism of the classical bound by introducing priorities to nodes within a DAG. They consider systems with a preemptive scheduler. With a given intra-task priority assignment, only nodes with higher priority need to be considered as the interference workload. Therefore, the resulting intra-task delay can be reduced. As shown in the *part 1* of Equation 27, C_{λ_j} calculates the workload of the path λ_j. The factor $I(\lambda_i)$ denotes the set of nodes that have a higher priority than the nodes in path λ_i. Term $C_{I(\lambda_i)}$ then calculates the interference workload of the path. Finally, $max_{\lambda_j \in \tau_i}\{\}$ iterates through all paths in τ_i and finds out the maximum bound on the DAG makespan. The *part 2* of Equation 27 denotes the inter-task delay, where only the interference from tasks with higher priorities is accounted for. The calculation of I_i^{hi} follows the same method as in Equation 25.

In Equation 27, the intra-task interference is reduced by assuming an intra-task priority assignment under the preemptive scheme. However, the parallelism between nodes is not fully exploited. For instance, the analysis assumes that all nodes with a higher priority contribute to intra-task interference when computing the worst-case makespan. Yet, when there are sufficient cores in the system, nodes can execute in parallel without experiencing any interference. Thus, it does not address Limitation 7. Furthermore, since the focus is on preemptive scheduling, it does not address Limitation 8.

$$R_i = \underbrace{\max_{\lambda_j \in \tau_i} \left\{ C_{\lambda_j} + \frac{C_{I(\lambda_j)}}{m} \right\}}_{\text{part 1}} + \underbrace{\left\lfloor \frac{I_i^{hi}}{m} \right\rfloor}_{\textbf{part 2}} \tag{27}$$

Later, Zhao et al. [107] proposed a RTA under the limited preemption scheme which aims to eliminate unnecessary interference by exploring the parallelism of the execution of nodes. The single-DAG analysis produces the worst-case makespan by computing the worst-case finish time of each node. According to Equations 3, 4, and 10 in [107], the key concept of calculating the worst-case finish time of each node can be summarized as Equation 28. In this analysis, the worst-case finish time of a node v_j is denoted as $Ft(v_j)$, which is computed by determining: 1) the WCET of v_j, 2) the predecessor node with the latest worst-case finish time, i.e. $\max_{v_k \in pre(v_j)}\{Ft(v_k)\}$, and 3) the intra-task delay which include the delay imposed by both interference and blocking (denoted as $\left\lceil \frac{C_{I(v_j)}}{m} \right\rceil$).

$$Ft(v_j) = C_{v_j} + \max_{v_k \in pre(v_j)}\{Ft(v_k)\} + \left\lceil \frac{C_{I(v_j)}}{m} \right\rceil \tag{28}$$

$$I(v_j) = S(v_j) \setminus \bigcup_{v_p \in anc(v_j)} I(v_p) \tag{29}$$

As illustrated in Equation 28, the set of nodes that can delay v_j is encompassed within the list $I(v_j)$. This list comprises both higher-priority nodes that can interfere with v_j and lower-priority nodes that can block v_j. The cumulative workload of the nodes in $I(v_j)$ is represented as $C_{I(v_j)}$.

Further examining the composition of $I(v_j)$ as depicted in Equation 29, it includes all the concurrent nodes of v_j, denoted as $S(v_j)$, excluding those that have already been accounted for in delaying the ancestors of v_j. This is achieved through a loop that iterates over each ancestor of v_j, i.e., $v_p \in \text{anc}(v_j)$, incorporating every list that has been recognized as containing nodes that can delay the ancestor v_p, represented as $I(v_p)$. However, this analysis might potentially be unsafe, a concern is proposed in Lemma 3.

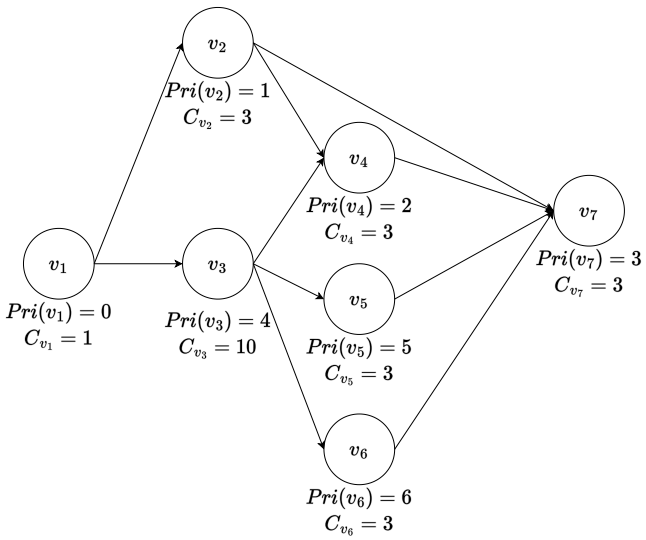


Figure 11: An example of a DAG.

Lemma 3. *The calculation of the worst-case finish time shown in Equation 28 is not necessarily the worst-case finish time.*

Proof. An example of a DAG task is shown in Figure 11. Assuming the DAG executes in a dual-core system. To ease the presentation, the scheduling scheme is assumed to be preemptive (i.e., no blocking delay can occur) which will not affect our claim. We now compute the worst-case finish time of each node and the makespan of the DAG using the analysis in [107].

According to Equation 28, we can calculate the worst-case finish time of each node as below:

- $Ft(v_1) = C_{v_1} = 1$: v_1 has no predecessors and concurrent nodes, so its finish time is equal to its computation time.
- $Ft(v_2) = C_{v_2} + Ft(v_1) + \frac{C_{v_3}+C_{v_5}+C_{v_6}}{2} = 3 + 1 + \frac{10+3+3}{2} = 12$. The only ancestor node of v_2 is v_1 and does not incur any interference. The concurrent nodes of v_2 are given by $S(v_2) = \{v_3, v_5, v_6\}$. These nodes have a higher priority than v_2.

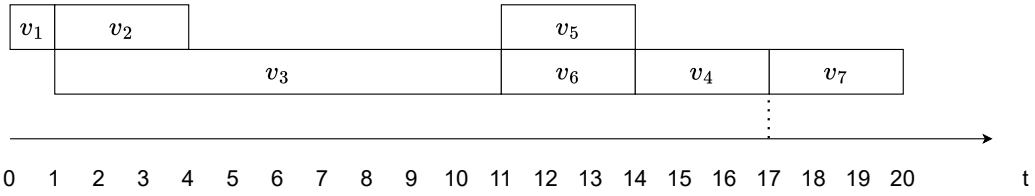


Figure 12: The simulation graph of the DAG in Figure 11.

- $Ft(v_3) = C_{v_3} + Ft(v_1) = 10 + 1 = 11$. Here, the only ancestor node of v_3 is v_1.The concurrent nodes of v_3 is $S(v_3) = \{v_2\}$, but $Pri(v_2) < Pri(v_3)$, so no interference is incurred.

- $Ft(v_4) = C_{v_4} + Ft(v_2) = 3 + 12 = 15$. In this case, $pre(v_4) = \{v_2, v_3\}$, and $\max_{v_k \in pre(v_4)}\{Ft(v_k)\} = Ft(v_2) = 12$. Since $S(v_4) = \{v_5, v_6\}$ has been accounted for in $Ft(v_2)$, hence no additional interference is calculated.

However, Figure 12 shows one of the possible execution scenario, in which v_5 and v_6 will delay the execution of v_4, which results in a finish time of $17 > 15$. However, such a delay is not accounted for in the analysis. Therefore, this analysis does not produce the worst-case finish time of the DAG. □

As for the analysis of multi-DAGs, the analysis in [107] does not support work-conserving scheduling, the inter-task delay cannot be computed using the same analytical approach as for the intra-task delay. It assumes that a DAG starts executing when the previous arrived DAG finishes executing entirely. Therefore, as the bound in [107] may not be safe, Limitations 7 and 8 are still open to be resolved, which motivates our work in the following chapter.

2.5.4 Summary

The section reviews research on DAG tasks in multi-core systems, emphasizing global work-conserving schedules. While the RTA of DAG tasks plays a central

role, various methods have been discussed, including generic DAG analysis and those with explicit intra-task priority. However, these methods often come with inherent limitations. These limitations include neglecting node parallelism, making pessimistic assumptions, and potential unsafety in certain analyses. Despite extensive research, these unresolved issues motivate the final step of the proposed HPRTS framework, as demonstrated in Chapter 5, to propose a new technique for RTA of DAG tasks and provide a tighter upper bound.

2.6 Summary of Existing Literature

In this chapter, an examination of real-time systems is presented. The focus is on their primary characteristics, the models used for tasks, and the scheduling mechanisms. The FPS method is highlighted due to its recognized efficiency and predictability within these systems. Furthermore, the RTA tool is identified as a vital instrument for evaluating whether tasks can be completed within their respective deadlines.

The subsequent section delves into the topic of resource sharing in real-time systems. The challenges of managing shared resources, especially in multicore environments, are discussed. Emphasis is placed on lock-based mechanisms, with spin-based locks being particularly noteworthy because of their role in ensuring mutual exclusivity of resource access. The importance of the FIFO order in managing access sequences is also addressed. A comparison is made between the older and newer versions of MSRP RTA, with the latter offering improvements in WCRT calculations.

The literature on MCS is explored, categorizing tasks based on their criticality levels. The classic AMC model is discussed for its approach to classifying tasks into different system modes. However, the four-mode model has certain limitations, summarized in Limitations 1, 2, and 3. These observations lead to

the proposition of advanced fault-tolerance models, as detailed in Chapter 3.

The topic of task allocation in real-time systems is addressed, highlighting a transition from static to dynamic methods. The SPA method, which groups tasks based on shared resources, is discussed, along with its potential to create large task groups. In contrast, the ROP method, which assigns tasks and resources separately, can lead to extended computation times. The limitations are summarized in Limitations 4, 5, and 6. These challenges and potential solutions are further elaborated upon in Chapter 4.

Finally, the chapter reviews DAG tasks in multi-core systems. The role of RTA in these tasks is central, but current methods have certain limitations. These limitations, including issues with node parallelism and specific assumptions, are summarized in Limitations 7 and 8. Moreover, we also spot a mistake of the state-of-the-art RTA as presented in [108] and highlight it in Lemma 3 with proofs. The need for a new RTA approach is emphasized, culminating in its introduction in Chapter 5.

Overall, this chapter provides a comprehensive review of real-time systems, focusing on reliable resource sharing, contention-aware task allocation algorithms, and RTA for DAG tasks. We then identify the bottlenecks in these areas that hinder the evolution of real-time systems. These limitations analyzed inspire our proposal of the HPRTS framework to guide modern real-time systems towards high performance

Chapter 3

Reliable Resource Sharing in Mixed-Criticality Systems

There is an increasing demand to realize both complex functionality and high performance with limited resources in emerging real-time applications. This necessitates extensive resource sharing. For example, to facilitate partially or fully automated driving, the AUTOSAR Classic standard (which implements static task configuration with resource isolation) is evolving to AUTOSAR Adaptive with dynamic resource sharing on multiprocessor architectures [5]. The conventional requirements of timing predictability and reliability still need to be satisfied. That is, the deadlines of tasks must be met while failures during task executions must be resolved. However, satisfying both is particularly hard, especially in the aerospace sector which makes extensive use of task re-execution to handle errors caused by radiation.

In this thesis, shared resources include data structures, special memory locations, and code segments. They often need to be accessed in a mutually exclusive fashion, which can cause blocking due to contention. Several multiprocessor resource-sharing protocols have been proposed to bound and minimize blocking time, including MSRP [41] and MrsP [22]. However, reliability has not been accounted for, which is imperative in mission-critical scenarios. Existing fault-tolerance methods are based on redundancy, and they may be directly applied to shared resources by scheduling repeated task executions and resource accesses a sufficient number of times to get the correct output. However, this leads to severe resource contention and undermines system schedulability.

This chapter consists the first step of the proposed HPRTS framework which

aims to bridge the gap between reliability and resource sharing. We focus on MCS that are widely found in practical applications [59, 101] and whose complicated execution model makes the problem more challenging to solve. We propose the first fault-tolerant solution for multiprocessor MCS with shared resources. First, a system execution model compatible with an arbitrary number of criticality levels is constructed. In our model, faults occurring in normal sections (i.e., without shared resources accessed) and critical sections (i.e., with shared resources accessed) are treated separately. Then, a novel protocol, namely Multiprocessor Stack Resource Protocol Fault Tolerance (MSRP-FT), is proposed to address faults during critical sections, aiming to minimize blocking time. Specifically, the remote processors occupied by other tasks waiting for the resource are employed to assist the resource-holding task (i.e., the head of the FIFO queue) for fault tolerance. A RTA is reported to bound the worst-case execution scenarios of the proposed solution.

The rest of this chapter is organized as follows: In Section 3.1, we present the proposed fault-tolerant solution for MCS with shared resources. In Section 3.2, we report the RTA for the proposed solution. Section 3.3 showcases the evaluation results, and Section 3.4 concludes the work and its success criteria established for the entire thesis.

3.1 A Fault-Tolerant Solution for MCS with Shared Resources

In this section, we present a new fault-tolerant solution for generic MCS that have tasks with two or more criticality levels with shared resources, to handle both task overruns and transient faults. First, we introduce a new fault-aware system model for MCS. The proposed system model distinguishes faults occurring in normal and critical sections, which enables different fault-tolerance schemes to

be implemented. Then, based on MSRP, a novel fault-tolerance multiprocessor resource sharing protocol is proposed for handling faults in critical sections, which reduces the blocking time incurred for tolerating faults and guaranteeing the reliability of the system.

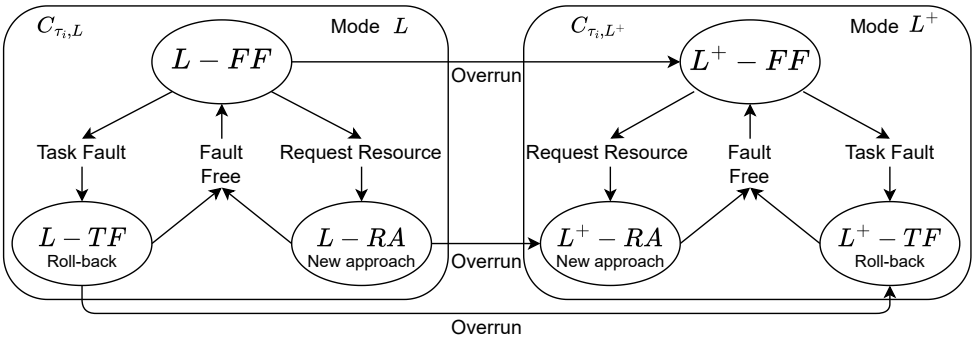


Figure 13: The proposed system model.

3.1.1 The Proposed System Model

To handle task overruns and faults which occur during both normal and critical sections of a MCS, a fault-tolerant system model is constructed based on the extension of the AMC model as illustrated in Section 2.3.2 [11]. Figure 13 illustrates the execution flow of the system and tasks in the model. During a task's execution, faults can occur either in a normal or a critical section. The former is called a *task fault* and the latter a *resource fault* in this chapter. In the proposed model, different fault-tolerant techniques are adopted to tolerate these two types of fault. The fault detection and tolerance techniques for normal and critical sections are presented in Section 3.1.2 and 3.1.3.

As shown in Figure 13, each task has three execution states under a system mode (say L): fault-free (L-FF), task-fault (L-TF) and resource-access (L-RA). They are allowed to execute up to an execution budget $C_{\tau_i,L}$. A task executing in state L-FF is executing a normal section without incurring any faults. Once

a fault occurs in a normal section, the task moves to state *L-TF*, at which the fault will be resolved. If a task requests a resource, it moves to state *L-RA* directly, where the fault-tolerance procedure for critical sections will be activated immediately, guaranteeing a fault-free resource access (see Section 3.1.3). The task moves back to state *L-FF* from *L-TF* or *L-RA* if the fault is resolved or the resource access is finished, respectively.

The system advances to the next system mode L^+ if any task with $l_i > L$ in mode L overruns its budget; this represents a transition between any two successive modes, e.g., from mode A to mode B. When an overrun occurs, tasks with criticality $l_i \geq L^+$ that are running in states *L-FF*, *L-TF* and *L-RA* will move directly to L^+*-FF*, L^+*-TF* and L^+*-RA* respectively with elevated execution budgets C_{τ_i,L^+} and other tasks are dropped. By doing so, each overrun can bring the system to the next mode, and therefore, the scalability issue stated in the Limitation 1 of Section 2.3.4 is addressed. However, there is an exception for tasks with criticality $l_i < L^+$ running in the state *L-RA* while executing with a shared resource, they are allowed to be dropped after finishing the underway critical section for the consideration of data integrity [47]. Moreover, mode changes can occur in the reverse direction. It is assumed that when the computational pressure decreases, the system reverts to the lowest mode, at which point previously suspended tasks are resumed. This functionality is facilitated through a dynamic monitoring system that evaluates the computational load in real time, thereby enabling fluid and efficient operation.

3.1.2 Fault-Tolerance of Normal Sections

In this subsection, we focus on transient faults which can be resolved by redundancy approaches. However, in systems with shared resources, detecting faults at the end of a task and re-executing the whole task to resolve a transient fault

can lead to substantial blocking time and the risk of transferring incorrect data to other tasks. To minimize the blocking time and provide reliable resource sharing, we apply different fault-tolerance approaches to handle faults that occur in normal and critical sections. This is achieved by not only inserting checkpoints at the start and end of each task but also introducing additional checkpoints around each critical section of the task. By doing so, the task execution is divided into a set of normal and critical sections. The acceptance test is assumed to be applied as the fault detection technique at each checkpoint. Although, the practical application of checkpointing in shared resource systems remains underexplored. This analysis advocates for its potential integration into future system architectures, highlighting its prospective benefits in supporting fault tolerance and operational efficiency. This proposition is a forward-thinking exploration aimed at guiding the development of more robust resource-sharing mechanisms in intricate computing environments.

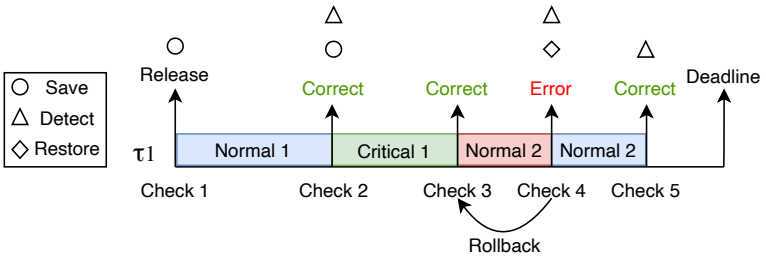


Figure 14: Fault-tolerance in normal sections

In the proposed fault-tolerance approach, the purposes of the checkpoints are slightly different, and so their operations vary. As shown in Figure 14, a checkpoint (*e.g.* Check 1) will be set at the beginning of a task to perform a *Save* operation which involves storing the current architectural state of the system, including register files, counter values and etc. For fault-tolerance in normal sections, each checkpoint will operate a *Detect* operation to detect faults after the execution of each normal segment. If no faults are detected (*e.g.* at Check 2)

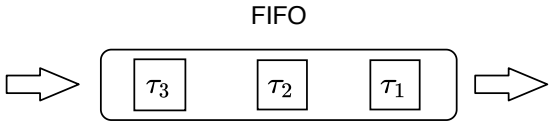
the checkpoint will perform the *Save* operation. Otherwise, if a fault is detected (*e.g.* at Check 4) the task will roll back to the most recent checkpoint and perform the *Restore* operation which restores the previous data and re-performs the execution. This process repeats until the normal section is executed without any fault. Each re-attempt requires an additional *Detect* operation (*e.g.* at Check 5). However, for the end of the last execution segment, the *Save* operation is not needed at the checkpoint.

3.1.3 Fault-Tolerance of Critical Sections by MSRP-FT

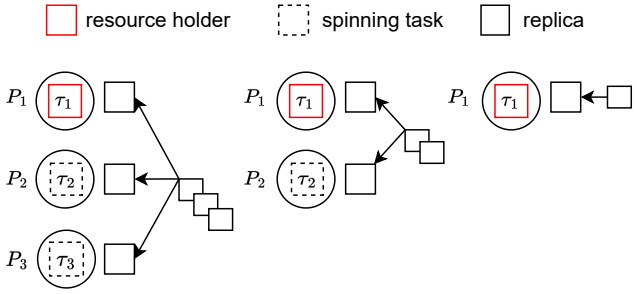
For faults occurring in critical sections, we propose a novel fault-tolerance multiprocessor resource sharing protocol, called MSRP-FT, in which tasks waiting for a resource can assist the resource holder to execute the associated critical section in parallel to address potential faults. The objective is to reduce the additional blocking time caused by resolving faults in critical sections via re-executions. The proposed MSRP-FT is introduced with the following steps.

Allocation of Replicas

Figure 15 demonstrates an example of the implementation of the proposed MSRP-FT, which is based on the resource sharing protocol MSRP. According to MSRP [41], tasks are inserted into a FIFO queue when they request a global resource. The task at the head of the queue (*e.g.* τ_1 in the figure) is granted the resource, other tasks spin on their own cores while checking the lock non-preemptively. With MSRP-FT, tasks are also placed at the FIFO queue when requesting shared resources. The task at the head of the FIFO queue will access the shared resource and the code segment to be executed by the head task and the internal states (*e.g.* variables) of the resource are replicated to a number according to the number of tasks in the FIFO queue as shown in Figure 15b. It is worth noting



(a) An example of a FIFO queue



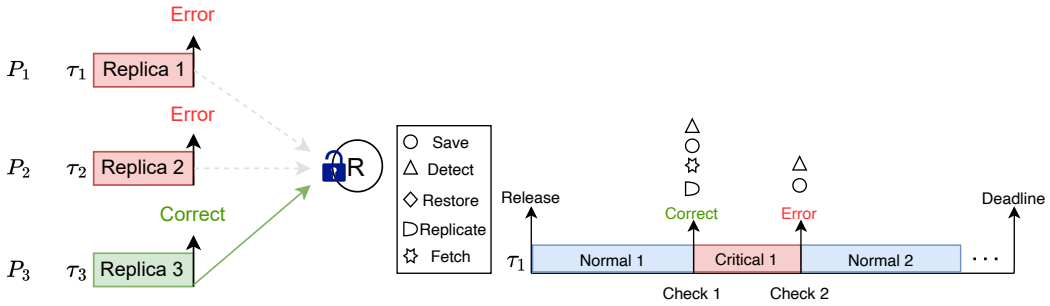
(b) Replicas allocation based on the number of tasks in the queue.

Figure 15: Fault tolerance in critical sections

that the access to the resource is always performed by the head task which obeys the mutually exclusive principle of shared resources and will not incur a race condition. Afterwards, replicas are stored in the local memory of each core and each task in the FIFO queue (including the head task) executes a replica on their host cores in parallel and updates the results on the local replica independently. Such approach allows critical sections to be executed by cores simultaneously without causing corruption. If there is only one task in the FIFO queue, the head task has to execute the critical section by itself.

Submission of Replicas

Each execution of the replica is tested for faults on different cores. As shown in Figure 16a, if a replica finishes without incurring any fault (*e.g.* on core P_3), it will obtain the lock and update the shared resource with its local variables. If two overlapping requests to acquire the lock arrive, one task will commit the result and another will have no effect on the resource. The update of the resource



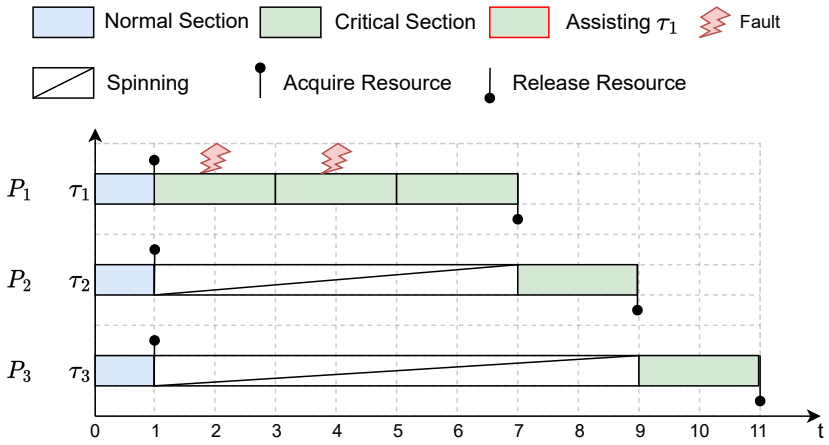
(a) Submission of execution results.

(b) Operations of checkpoints around a critical section.

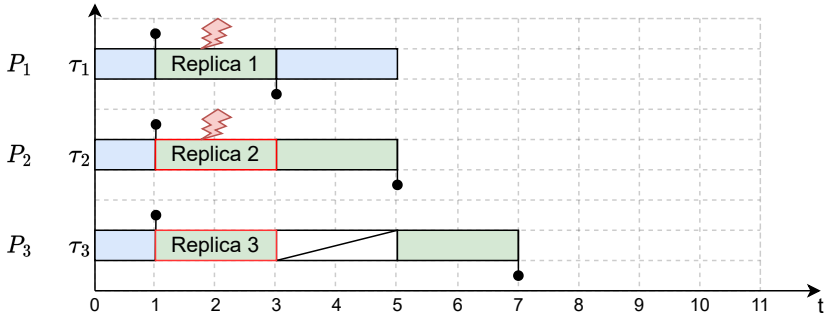
Figure 16: Fault-tolerance in a critical section

is assumed to be conducted with an atomic action which once performed no other action can interleave with it, hence, race conditions are avoided. Once the resource is updated, other tasks are signaled to abandon the computation. In contrast, if all the resource-accessing tasks fail to obtain the correct result, they roll back and re-execute the replica until the correct result is successfully submitted. With a successful commit by any task in the FIFO queue, the head task (i.e., τ_1) is removed from the queue and continues its execution. The same procedure then repeats for the next head task within the FIFO queue.

Figure 16b shows the operations performed at the checkpoints around the critical section of τ_1. The checkpoint at the start of the critical section (*e.g.* Check 1) first performs *Detect* and *Save* operations to detect for faults and save the results of the execution of the previous segment, which is the same as mentioned above. It also applies *Fetch* and *Replicate* operations to fetch and replicate the corresponding operation and the shared resources to the spinning cores. A *Detect* operation is performed after the execution of the replica. Although the replica incurs faults, τ_3 already updated the result and a *Save* operation is performed to save the architectural states of the system and τ_1 continues its execution. As demonstrated, the proposed approach abbreviates the delaying time due to fault-



(a) Fault tolerance by simple segment re-execution.



(b) The proposed fault-tolerance method.

Figure 17: A comparison between two fault-tolerance approaches under the same checkpoints setting.

tolerance and the submission made to the global resource is always guaranteed to be correct, therefore, both Limitations 2 and 3 introduced in Section 2.3.4 are addressed.

Working Example

To clarify the implementation of the proposed fault-tolerance approach, the detailed execution procedure of the example stated above under two different fault-

tolerance approaches is presented in Figure 17. Figure 17a assumes that each critical section is checked for faults and any detected fault is tolerated directly by the roll-back and re-execution approach. As shown in Figure 17a, τ_1, τ_2 and τ_3 request for a shared resource concurrently at $t = 1$. According to MSRP, τ_1 ranks first in the FIFO queue so it is granted with the resource and starts to execute its critical section immediately. Other tasks (τ_2 and τ_3) spin on their own cores and wait for the resource. However, τ_1 incurs two faults consecutively and re-executes its critical section twice. It finally releases the resource and leaves the FIFO queue at $t = 7$. τ_2 then becomes the head of the queue, which acquires the resource and starts its critical section from then.

With the application of the proposed fault-tolerant approach, as shown in the Figure 17b, the cores of τ_2 and τ_3 are utilized to execute $\tau_1's$ critical section in parallel instead of spinning. Although only one piece of the replica (i.e., Replica 3) is executed without faults, τ_1 can still continue its execution at $t = 3$. The chief principle of the fault-tolerant approach for critical sections is to replace wasted cycles of the spinning tasks in the FIFO queue to provide the reliability guarantee for each critical section in a single access, in pursuance of reducing the time spent on fault-tolerance and resource contention. For local resources, each task has to execute by itself as there exists no spinning tasks on remote cores.

Implementation and Run-Time Overhead

The implementation of the proposed approach requires the hardware architecture to have individual cache memory or dedicated memory space for each core to store replicas during the execution of the MSRP-FT, where most commercially off-the-shelf architectures can satisfy. From the software aspect, a global scheduler will be adopted to communicate with tasks on different cores. For example, the scheduler will signal tasks to assist the head task (*i.e.* the resource holder)

to execute the replicas in parallel. Once a successful result is submitted, the scheduler will signal other tasks to abandon the execution on replicas. Threads control methods such as *wait()* and *notify()* can be used to construct the above communication logic.

The feasibility of a task executing operations on behalf of other tasks has been validated in [91], in which once a task is preempted while spinning in the FIFO queue, the task behind it can acquire the lock first and execute the operation on behalf of the preempted task. Burns and Wellings [22] also briefly describes how the associated computations of the preempted task holder can be executed by the spinning tasks in parallel on different cores, but a detailed system design and implementation execution framework are not provided. Although the proposed fault-tolerance approach is developed within a different context and serves a different purpose, that of reducing blocking time caused by resource faults, the above work has provided sufficient evidence towards the applicability and practicability of the proposed approach.

Moreover, the setting of checkpoints can bring additional overheads in terms of execution time. However, there is a clear trade-off between the number of checkpoints being set and the final schedulability benefits of the proposed approach. If the task has intensive resource requests (i.e. contains voluminous critical sections), the engineer can set fewer checkpoints in a flexible manner so that a balanced result can still be achieved between the time spent for each checkpoint and the advantage brought by the proposed approach. In this work, we assume the worst-case overhead (including that of checkpoints and communication) is included in the WCET of each task.

3.2 Schedulability Analysis

This section provides the RTA for MCS with the proposed fault-tolerance solution, in which resources are managed by MSRP-FT. The RTA is based on the most state-of-the-art analysis of MSRP [109] and the AMC-rtb analysis of the AMC model [11] as stated in Section 2.2.4 and 2.3.2 respectively. Our novelty is developing the WCRT bound of the fault-tolerant approaches introduced in Section 3.1 and integrating these analyses together to provide a holistic upper bound for the target systems under the proposed approach. We present the complete worst-case RTA incrementally with an increasing complexity. First, the analysis for systems that only incur transient faults is developed, i.e. in a stable system mode. Then, we extend the analysis to include overruns which will advance the system mode from L to L^+, i.e. a mode switch. In addition, this analysis does not depend on a specific fault model (e.g. the one in [83]). Similar with [1], each task τ_i is assumed to incur an individual number of faults $f_{i,L}$ during one execution in a mode L, in the worst case. Computing the number of faults that a task can tolerate is referred to as the fault model [1, 83, 84] but is out of the scope of this thesis.

3.2.1 Analysis of Systems with A Stable Mode

Equation (30) bounds the WCRT of a task τ_i in a stable mode L. As shown in the equation, $C_{\tau_i,L}$ denotes total execution time of normal sections of τ_i in mode L. Notation E_i^L represents the resource accessing time incurred by τ_i and the set of tasks on the same core but that have priority higher than τ_i and with criticality $l_i \geq L$ (i.e., $lhpH(i, L)$). B_i^L represents the arrival blocking incurred by τ_i in mode L. The resource accessing time is the pure computation time of a task with shared resources and the spin delay experienced by the task when requesting shared resources but being delayed by other tasks in the FIFO queue which

arrived earlier. Whereas, the arrival blocking is incurred when the task is released but blocked by local low priority tasks that are accessing shared resources and cannot be preempted. The interference is bounded in $\sum_{j \in \mathbf{lhpH}(i,L)} \left\lceil \frac{R_i^L}{T_j} \right\rceil \times C_{\tau_j,L}$, which is the total amount of execution time (normal sections only) preempted by $lhpH(i,L)$. F_i^L and F_j^L denote the additional execution time spent on recovering faults occurred on τ_i and a task τ_j in $lhpH(i,L)$ respectively.

$$\begin{aligned} R_i^L = & C_{\tau_i,L} + E_i^L + B_i^L + \sum_{j \in \mathbf{lhpH}(i,L)} \left\lceil \frac{R_i^L}{T_j} \right\rceil \times C_{\tau_j,L} \\ & + F_i^L + \sum_{j \in \mathbf{lhpH}(i,L)} \left\lceil \frac{R_i^L}{T_j} \right\rceil \times F_j^L \end{aligned} \tag{30}$$

Resource Accessing Time (E_i^L)

As introduced in Section 2.2.4, the spin delay is the blocking incurred by τ_i when:

- requests a shared resource but is blocked directly by tasks running on remote cores accessing the same resource; or
- is preempted by a local higher priority task ($lhpH(i,L)$), which in turn, is blocked directly by remote tasks for accessing a shared resource.

However, it is not always true that each request sent by τ_i and $lhpH(i,L)$ will be blocked by accesses from every remote core that contains tasks requesting the same resource as illustrated in the Example 4. This observation aligns with the state-of-the-art analysis of MSRP presented in [104]. Additionally, in our model, each request to the shared resource may execute more than once due to the occurrence of faults, since each execution with shared resource (i.e., critical section) is tested for transient faults and handled immediately with the proposed MSRP-FT. Moreover, in MSRP-FT, the spinning tasks will also execute the resource along with the head task in parallel to hasten the fault-tolerance process. Therefore, the worst-case execution time of the proposed approach needs to be

developed, and we claim novelty of the following parts of the analysis. Notations introduced for bounding the resource accessing time are presented in Table 8.

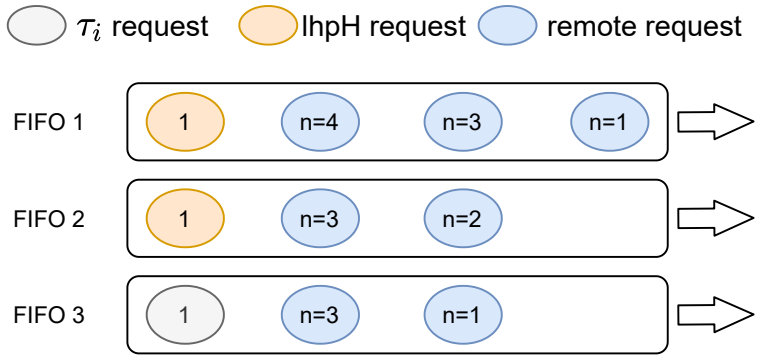
The maximum number of executions for each request to resource r^x is characterized as $n^x_{i,L} = 1 + f_{i,L}$, which includes the successful one, and a series of failed executions $f_{i,L}$ (i.e., number of faults). Lemma 4 is then stated to provide the worst-case delay that a task can incur from the resource holder during its one request to a resource.

Lemma 4. *For task τ_i that is spinning for r^x at position k in the FIFO queue, the worst-case delay τ_i that can be incurred from the current resource holder τ_j is bounded by $\left\lceil \frac{n^x_{j,L}}{k} \right\rceil c^x$.*

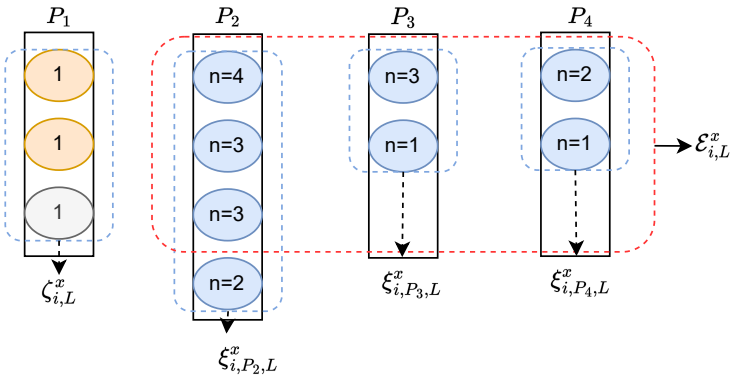
Proof. Follows directly from the properties in the fault-tolerance method for global resources. In the worst case, τ_i and all $(k-1)$ tasks ahead τ_i (including τ_j) will execute r^x in parallel, and the maximum execution number of the request is $n^x_{j,L}$ times. Hence, the maximum length of τ_j's access to r^x (also the delay to τ_i) is bounded by $\left\lceil \frac{n^x_{j,L}}{k} \right\rceil c^x$. □

According to Lemma 4, as shown in Figure 18a, in the worst case, each request issued from τ_i's core endures every possible blocking and is placed at the end of the FIFO queue, with no help available from other waiting tasks to resolve faults. Moreover, remote blocking requests with higher $n^x_{i,L}$ should be placed closer to the end of the FIFO queue (i.e. fewer helpers and smaller k) to impose the worst delay to τ_i.

To express the worst-case scenario with formulations, we introduce $\eta^x_{i,L}(\iota, \mu)$ to denote a non-increasing list of requests from a task τ_i to resource r^x during a time period ι with jitter μ. Each element in the list is the number $n^x_{i,L}$. It follows that $|\eta^x_{i,L}(\iota, \mu)| = \left\lceil \frac{\iota+\mu}{T_i} \right\rceil \times N^x_i$, where $||$ gives the size of the list (i.e., the number of requests) and N^x_i is the number of τ_i's accesses to r^x in one release.



(a) Worst-case placement of remote blocking requests



(b) Requests to a resource from different processors

Figure 18: Worst-case spin delay

For instance, $\eta^x_{i,L}(R_i, 0) = \{2, 2, 2\}$ gives $N^x_i = 3$ and all $n^x_{i,L} = 2$[1].

$$\zeta^x_{i,L} = \eta^x_{i,L}(R^L_i, 0) \cup \bigcup_{\tau_j \in \mathbf{lhpH}(i,L)} \eta^x_{j,L}(R^L_i, 0) \tag{31}$$

With $\eta^x_{i,L}(\iota, \mu)$ defined, we introduce a set of notations of lists as shown in Figure 18b for bounding the blocking, where P_1 is the local core and $P_2 - P_4$ are remote cores. Below we explain each of the notations introduced in the figure and their formulations. Equation 31 gives the non-increasing list of execution

[1]In $\eta^x_{i,L}(\iota, \mu)$, each execution number also denotes one resource request. For simplicity, we use these two terms interchangeably.

Table 8: Notations for Bounding the Resource Accessing Time

Symbols	**Definitions**
$n^x_{i,L}$	The number of executions of a request from τ_i to r^x in mode L.
$f_{i,L}$	The maximum number of faults that τ_i can incur in mode L.
N^x_i	The number of requests from τ_i to resource r^x during one release.
$\eta^x_{i,L}(\iota, \mu)$	A list of requests from τ_i to r^x during period ι with a jitter μ in mode L.
$lhpH(i, L)$	The list of local high-priority tasks that are eligible to execute in mode L.
$\zeta^x_{i,L}$	The list of execution numbers of each access to r^x by τ_i and $lhpH(i, L)$ in mode L.
$\xi^x_{i,P_a,L}$	The list of execution numbers of each remote request that can block τ_i or $lhpH(i, L)$ from core P_a in mode L.
$\mathcal{E}^x_{i,L}$	The list of remote requests that can block τ_i or tasks in $lhpH(i, L)$ from all remote cores in mode L.
$\Gamma(P_a, L)$	The set of tasks on a remote core with index P_a in mode L.

numbers of each access to r^x by τ_i and tasks in $lhpH(i, L)$, during the release of τ_i. If a list is denoted by $\zeta^x_{i,L}$, then $\zeta^x_{i,L}(k)$ gives its kth element, or $\varnothing$ if $k > |\zeta^x_{i,L}|$. Note, $\zeta^x_{i,L}$ does not account for the requests issued by τ_i and tasks in $lhpH(i, L)$ for addressing faults through re-execution. The additional time caused by this is bounded in F^L_i and F^L_j, in Section 3.2.1.

In addition, a remote task τ_j can incur at most $f_{j,L}$ faults during its execution while accessing shared resources. Accordingly, the worst-case blocking for τ_i happens when all $f_{j,L}$ faults occur in its critical sections that can block either τ_i or tasks in $lhpH(i, L)$. However, τ_j can have multiple requests to different resources in one release, where all these requests can block τ_i in the worst-case. Moreover, among these requests, the one for the longest resource may not impose the worst spin delay to τ_i according to Lemma 4. For instance, a task τ_j is executing with a resource r^x up to $n^x_{j,L} = 5$ times with $c^x = 5$, and τ_i's request is placed in the FIFO queue with $k = 2$. Thus, the worst spin delay τ_j can impose to τ_i is

$\lceil \frac{5}{2} \rceil * 5 = 15$. If $c^x = 10$ and τ_i is placed at $k = 10$, then the spin delay τ_j can impose to τ_i is $\lceil \frac{5}{10} \rceil * 10 = 10$. However, the identification of the remote request that can cause the worst-case blocking with faults would require significant state exploration. Therefore, to ease computation while providing a safe bound, we assume the first request of τ_j to each resource that can block τ_i will incur $f_{j,L}$ faults (i.e., $n^x_{i,L} = 1 + f_{j,L}$). As each task can incur at most $f_{j,L}$ faults during real execution in the worst-case, this assumption does not jeopardize the safe bound as a higher number of faults during a resource access can only lead to a non-decreasing blocking.

The notation $\xi^x_{i,P_a,L}$ is then introduced as the non-increasing list of remote requests that can block τ_i or tasks in $lhpH(i, L)$ from the core with index a. $\Gamma(P_a, L)$ is the set of tasks on a remote core P_a in mode L, $\xi^x_{i,P_a,L}$ loops though every task on core P_a and combines their $\eta^x_{j,L}(R^L_i, R^L_j)$ lists with the inclusion of back-to-back hit (see Section 2.2.4) together to form a new list, examples can be seen in Figure 18b.

$$\xi^x_{i,P_a,L} = \bigcup_{\tau_j \in \Gamma(P_a,L)} \eta^x_{j,L}(R^L_i, R^L_j) \tag{32}$$

With Equations 31 and 32, the $\mathcal{E}^x_{i,L}$ is introduced as a non-increasing list to store remote requests that can cause τ_i to incur the worst-case spin delay and can be obtained as given in Equation 33. This equation iterates through each remote core P_a and takes the first $|\zeta^x_{i,L}|$ elements in $\xi^x_{i,P_a,L}$ (if they exist), i.e. at most $|\zeta^x_{i,L}|$ of requests from each core can block τ_i or $lhpH(i, L)$ as mentioned above. Provided that all lists are in non-increasing order, hence the top elements are with highest $n^x_{i,L}$ value.

$$\mathcal{E}^x_{i,L} = \bigcup_{P_a \neq P(\tau_i)} \bigcup_{k=1}^{|\zeta^x_{i,L}|} \xi^x_{i,P_a,L}(k) \tag{33}$$

Finally, E^L_i can be bounded as Equation 34. First, for $r^x \in \mathbb{R}$, each request in $\zeta^x_{i,L}$ accounts for one execution and takes $1 \times c^x$ units of time to finish.

Accordingly, the total execution time of all requests in $\zeta^x_{i,L}$ is $|\zeta^x_{i,L}|\times c^x$. Second, for requests in $\mathcal{E}^x_{i,L}$, the delay is bounded based on the principle that *a request with a larger number of total executions receives less help*. Thus, given $|\zeta^x_{i,L}|$ (i.e., total number of FIFO queues), the kth request in $\mathcal{E}^x_{i,L}$ can have at most $\lceil k/|\zeta^x_{i,L}|\rceil$ helping tasks in one FIFO queue, and hence, a finish time of $\lceil \mathcal{E}^x_{i,L}(k)/(\lceil k/|\zeta^x_{i,L}|\rceil + 1)\rceil \times c^x$. Finally, we note that no delay is imposed to τ_i from r^x if $\zeta^x_{i,L} = \varnothing$; and that $|\mathcal{E}^x_{i,L}|= 0$ if r^x is a local resource (no blocking from remote tasks).

$$E^L_i = \sum_{r^x \in \mathbb{R}} \left(|\zeta^x_{i,L}| + \sum_{k=1}^{|\mathcal{E}^x_{i,L}|} \left\lceil \frac{\mathcal{E}^x_{i,L}(k)}{\left\lceil k/|\zeta^x_{i,L}| \right\rceil + 1} \right\rceil \right) \times c^x \tag{34}$$

Recall Figure 18 for instance, τ_i and tasks in $lhpH(i, L)$ request for r^x three times in total, i.e., $|\zeta^x_{i,L}|= 3$, where these three requests can be blocked by a list of remote requests $\mathcal{E}^x_{i,L} = \{4, 3, 3, 3, 2, 1, 1\}$. With $c^x = 1$, the delay is bounded by $(\lceil 4/2\rceil + \lceil 3/2\rceil + \lceil 3/2\rceil + \lceil 3/3\rceil + \lceil 2/3\rceil + \lceil 1/3\rceil + \lceil 1/4\rceil) \times 1 = 10$.

Theorem 1. *Equation 34 bounds the worst-case spin delay of τ_i in system mode L.*

Proof. First, the lists of requests to r^x from τ_i's core and all remote cores that can block τ_i is bounded by $\zeta^x_{i,L}$ and $\mathcal{E}^x_{i,L}$, respectively. This is proved by Theorems 1 and 2 in [109].

Second, Equation 34 accounts for the worst-case delay from requests in $\mathcal{E}^x_{i,L}$. We prove this by swapping any two elements in $\mathcal{E}^x_{i,L}$ with indexes k_1 and k_2 and values n_1 and n_2, respectively. Let $\varepsilon_j = \lceil k_j/|\zeta^x_{i,L}|\rceil + 1$, it follows $n_1 \geq n_2$ and $\varepsilon_1 \leq \varepsilon_2$ if $k_1 < k_2$. Accordingly, we have $\lceil n_1/\varepsilon_1\rceil + \lceil n_2/\varepsilon_2\rceil$ and $\lceil n_2/\varepsilon_1\rceil + \lceil n_1/\varepsilon_2\rceil$ before and after the swap, respectively. It is clear that $(n_1/\varepsilon_1 + n_2/\varepsilon_2) \geq (n_2/\varepsilon_1 + n_1/\varepsilon_2)$ as $n_1 \times \varepsilon_2 + n_2 \times \varepsilon_1 - n_1 \times \varepsilon_1 - n_2 \times \varepsilon_2 = (n_1 - n_2) \times (\varepsilon_2 - \varepsilon_1) \geq 0$. Therefore, the theorem holds. □

Arrival Blocking (B_i^L)

$$B_i^L = \max\{(\alpha_{i,L}^x + \sum_{k=1}^{|\beta_{i,L}^x|} \left\lceil \frac{\beta_{i,L}^x(k)}{k+1} \right\rceil) \times c^x | r^x \in F^A(i,L)\} \tag{35}$$

Arrival blocking B_i^L is given by Equation 35, in which $F^A(i,L)$ gives the set of resources that can block τ_i upon its arrival (global or local resources with a ceiling higher than τ_i's priority) that are accessed by τ_i's local low priority tasks ($llpH(i,L)$). As arrival blocking can only occur once before τ_i's execution, it is bounded by computing the largest delay for accessing a resource in $F^A(i,L)$ from a task in $llpH(i,L)$ [109].

Function $\alpha_{i,L}^x = \max\{n_{j,L}^x | \tau_j \in llpH(i,L) \wedge N_j^x > 0\}$ gives the request with the highest total execution number issued by $llpH(i,L)$ to r^x. Function $\beta_{i,L}^x = \bigcup_{P_a \neq P_{\tau_i}} \xi_{i,P_a,L}^x(|\zeta_{i,L}^x|+1)$ gives the requests with the highest execution number in each remote core that can block $\alpha_{i,L}^x$. For a remote core P_a, the request that can cause the highest blocking is $\xi_{i,P_a,L}^x(|\zeta_{i,L}^x|+1)$ (if any), in which the requests before index $|\zeta_{i,L}^x|+1$ have been accounted for in Equation 34 when bounding τ_i's spin delay. $\beta_{i,L}^x = \varnothing$ if r^x is a local resource.

With the above functions, the worst-case resource accessing time of a task in $llpH(i,L)$ can be obtained by the same way as Equation 34. For request $\alpha_{i,L}^x$, it takes $\alpha_{i,L}^x \times c^x$ to finish its execution with r^x (i.e., at the end of the FIFO queue) and incurs a worst-case blocking of $\sum_{k=1}^{|\beta_{i,L}^x|} \left\lceil \frac{\beta_{i,L}^x(k)}{k+1} \right\rceil \times c^x$ from requests in $\beta_{i,L}^x$, with the proposed fault-tolerance method applied.

Local Fault-Tolerance Time (F_i^L, F_j^L)

The proposed fault-tolerance approach divides tasks into several segments, the worst-case fault-tolerance scenario happens when all $f_{i,L}$ faults happen in the longest segment of a task repeatedly [80]. The additional execution time of τ_i and tasks in $lhpH(i,L)$ due to fault-tolerance is bounded in Equation 36, in

which the longest segment is selected among the normal section $C_{\tau_i,L}$ and a set of critical section lengths c^x from the set of resources accessed by τ_i in mode L (denoted as $F(\tau_i, L)$). If the longest segment is one of critical section c^x, the external time spent on fault-tolerance is $f_{i,L} \times c^x$, as all requests from τ_i and $lhpH(i, L)$ are executed without helpers.

$$F_i^L = f_{i,L} \times \max\{C_{\tau_i,L}, \max\{c^x | r^x \in F(\tau_i, L)\}\} \tag{36}$$

3.2.2 Analysis of Systems under A Mode Switch

If any task overruns its execution budget in mode L, the system would switch to a higher mode (say L^+). During the mode switch, tasks with criticality $l_i < L^+$ are dropped and ones with criticality $l_i \geq L^+$ are granted higher execution budgets. The worst-case response time for a task τ_i that experiences faults and a mode switch from L to L^+ is given by Equation 37, in which $lhpE(i, L)$ and $lhpH(i, L^+)$ represent the set of local high priority tasks with a criticality level $l_i = L$ and $l_i \geq L^+$ respectively.

$$\begin{aligned} R_{i,L}^{L^+} =& C_{i,L^+} + E_{i,L}^{L^+} + B_{i,L}^{L^+} + F_i^{L^+} \\ &+ \sum_{j \in \mathbf{lhpH}(i,L^+)} \left\lceil \frac{R_{i,L}^{L^+}}{T_j} \right\rceil C_{j,L^+} \\ &+ \sum_{j \in \mathbf{lhpE}(i,L)} \left\lceil \frac{R_i^L}{T_j} \right\rceil C_{j,L} \\ &+ \sum_{j \in \mathbf{lhpH}(i,L^+)} \left\lceil \frac{R_{i,L}^{L^+}}{T_j} \right\rceil \times F_j^{L^+} \\ &+ \sum_{j \in \mathbf{lhpE}(i,L)} \left\lceil \frac{R_i^L}{T_j} \right\rceil \times F_j^L \end{aligned} \tag{37}$$

The blocking terms $E_{i,L}^{L^+}$ and $B_{i,L}^{L^+}$ are bounded by the same approaches in Equations 31 to 35, with the following two changes. First, the system mode

is updated from L to L^+ in all equations. This reflects the increased busy-period, i.e., $R_{i,L}^{L^+}$ and $R_{j,L}^{L^+}$ in Equations 31 and 32; and the set of tasks that are active in the new mode L^+, i.e., $lhpH(i, L^+)$, $H(P_a, L^+)$ and $llpH(i, L^+)$ used by Equations 31, 32 and 35. Second, when accounting for remote requests in 32, tasks with criticality $l_i \geq L^+$ can incur at most f_{i,L^+} faults, where $f_{i,L^+} \geq f_{i,L}$, since longer execution time has a higher probability of incurring faults [1]. The terminating tasks (tasks with a criticality level L) are included when accounting for resource requests in Equations 31, 32 and 35, with a busy period of R_i^L and maximum fault $f_{i,L}$ (where applicable).

In addition, according to the AMC-rtb analysis [11], the worst-case scenario of τ_i during a mode switch happens when: 1) all tasks in $lhpE(i, L)$ preempt τ_i and finish executing with $C_{\tau_i,L}$ during the busy period R_i^L before being terminated; 2) all tasks in $lhpH(i, L^+)$ preempt τ_i and execute with C_{τ_i,L^+} during the period $R_{i,L}^{L^+}$ (with $R_{i,L}^{L^+} > R_i^L$). The theories are reflected in the second and third lines of Equation 37, in which the interference from $lhpH(i, L^+)$ and $lhpE(i, L)$ last for the busy periods $R_{i,L}^{L^+}$, and R_i^L respectively.

Finally, the fault-tolerance time of τ_i and tasks in $lhpH(i, L^+)$ and $lhpE(i, L)$ are denoted as $F_i^{L^+}$, $F_j^{L^+}$ and F_j^L respectively. They are bounded in the same way as Equation 36, but for $F_i^{L^+}$ and $F_j^{L^+}$, mode L is changed to L^+ correspondingly.

3.3 Evaluation

In this section, the schedulability of the *proposed* model is investigated against the state-of-the-art *Four-mode* model [1] and the *Checkpointing* model. The RTA in Section 3.2 implements the following comparisons:

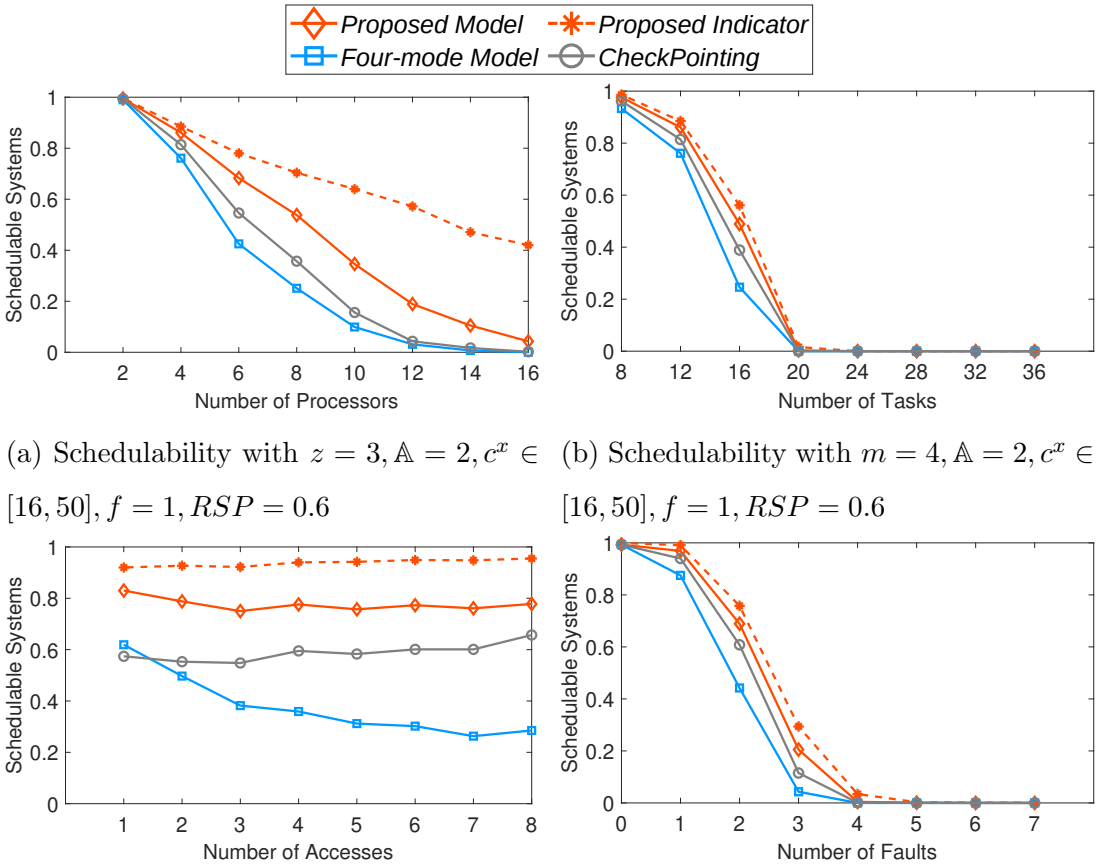
- the *Proposed Model*;
- the *Four-mode Model*: once a fault is detected at the end of the task, the whole task is re-executed and shared resources are re-requested;

- the *Checkpointing Model*: Same checkpoint setting as the proposed model, but faults are resolved by simple segment re-execution (see Figure 17).

- the *Indicators* of the *proposed* model showing the best-case resource-access scenario where tasks incur no delay when accessing resources. This is used to indicate scheduling pressure of the system. (*e.g.* $E_i^L = \sum_{r^x \in \mathbb{R}} |\zeta_{i,L}^x| \times c^x$ and $B_i^L = 0$ under mode L).

Experimental setup: To clarify presentation, we consider systems that have tasks with two criticality levels ($l_i \in \{low, high\}$) and systems have two modes $L = LO$ and $L^+ = HI$. Tasks are allocated by the Worst-Fit (WF) heuristic and are scheduled by fully-partitioned preemptive Fixed-Priority scheduling (FPS). Rate Monotonic Priority Assignment (RMPA) is applied. We consider platforms with a set of processors $m \in [2, 16]$ and a set of tasks on each processor $z \in [2, 9]$ which enables a proper range of schedulable systems for analysis and comparison. The low criticality utilization of each task ($U_{\tau_i,low}$) is generated using the UUnifast algorithm [14], with a system utilization bound $U_{\tau_i,low} = 0.1 \times m \times z$. Task periods are randomly selected between [1ms,1000ms] in a log-uniform distribution. Deadlines of the tasks are equal to their periods. For a task τ_i, the low-criticality worst-case execution time for normal sections is given by $C_{\tau_i,low} = U_{\tau_i,low} \times T_i - \sum_{r^x \in \mathbb{R}} N_i^x \times c^x$. For each system, half of the tasks (chosen randomly) are assigned a high criticality, where $C_{\tau_i,high}$ is randomly set to $[1, 1.5]$ times of its $C_{\tau_i,low}$.

With an increasing number of processors, we introduce a corresponding increase in the number of shared resources. Therefore, the number of shared resources is set to equal m. Shared resources are managed by $MSRP$ but incorporated with different fault-tolerance approaches. Among the generated tasks, a ratio of tasks are selected to request a random number of resources (up to m). The ratio is denoted the Resource-Sharing Percentage (RSP). The number of



(a) Schedulability with $z = 3, \mathbb{A} = 2, c^x \in [16, 50], f = 1, RSP = 0.6$

(b) Schedulability with $m = 4, \mathbb{A} = 2, c^x \in [16, 50], f = 1, RSP = 0.6$

(c) Schedulability with $m = 4, z = 3, c^x \in [51, 100], f = 1, RSP = 0.8$

(d) Schedulability with $m = 4, z = 2, \mathbb{A} = 3, c^x \in [16, 50], RSP = 0.8$

Figure 19: LO mode

accesses to a resource from each task is set to $\mathbb{A} \in [1, 8]$ in one release. The worst-case computation time of a shared resource (c^x) is generated randomly in the range $[1\mu s, 100\mu s]$. With c^x obtained, $C_{\tau_i,low}$ described above can be computed, where we enforce $U_{\tau_i,low} \times T_i - \sum_{r^x \in \mathbb{R}} N_i^x \times c^x \geq 0$.

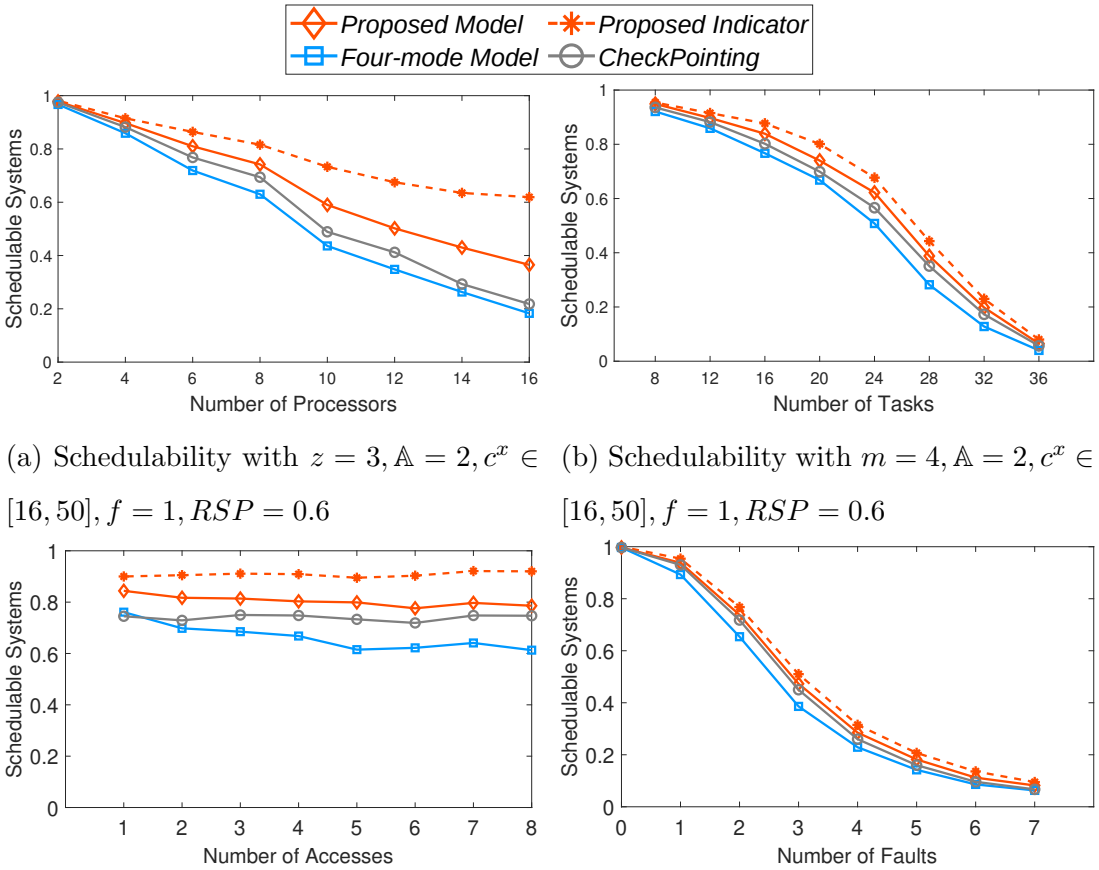
To enable fair comparisons and highlight the schedulability differences between the *proposed*, *Four-mode*, and *Checkpointing* models, we assign each task to incur the same number of faults f for all models, with $f \in [0, 7]$.

Results: Figures 19, 20 and 21 illustrate the schedulability of systems at

LO mode, HI mode, and during a Mode Switch, respectively. In each figure we vary system settings m, $m \times z$ (number of tasks in the system), $\mathbb{A}$ and f. In addition, Figure 22 shows the schedulability of all modes with the variation of RSP. For each combination of system settings, 1000 systems are generated and the percentage of schedulable systems of the considered models is presented.

Figures 19 and 20 present the schedulability of the evaluated models under stable modes LO and HI, respectively. From the results, the schedulability of the *proposed* model outperforms the *Four-mode* and the *Checkpointing* models in all considered settings. In Figures 19a and 20a (with varied m), the schedulability of all models decreases with the increase of m, due to increased remote blocking time. However, the *proposed* model maintains an advantage over the *Four-mode* and the *Checkpointing* models in terms of schedulability. The results are due to the setting of checkpoints allowing the fault-tolerance approach to be implemented on small task segments, and the *proposed* model effectively reduces remote blocking by allowing spinning tasks to execute work for the resource holder in parallel, instead of re-requesting resources when the whole task is re-executed or re-executing the resource a number of times sequentially.

For the same reason, the *proposed* model provides better schedulability than the *Four-mode* and the *Checkpointing* models in Figures 19c and 20c with varied $\mathbb{A}$. With the *proposed* model, the increasing $\mathbb{A}$ can bring more critical sections to a task, thus with more checkpoints, which allows the fault-tolerance to be carried out on smaller segments. As shown in Figure 19c, for instance, with $\mathbb{A} = 7$, the *proposed* model can schedule 189.4% more systems than the *Four-mode* model, where the *Four-mode* model schedules 263 systems and the *proposed* model can schedule 761 systems. On the other hand, under the same setting of checkpoints, the *proposed* model also outperforms the *Checkpointing* model by 13.3 % with $\mathbb{A} = 1$ in Figure 20c. Although the increased number of checkpoints will bring



(a) Schedulability with $z = 3, \mathbb{A} = 2, c^x \in [16, 50], f = 1, RSP = 0.6$

(b) Schedulability with $m = 4, \mathbb{A} = 2, c^x \in [16, 50], f = 1, RSP = 0.6$

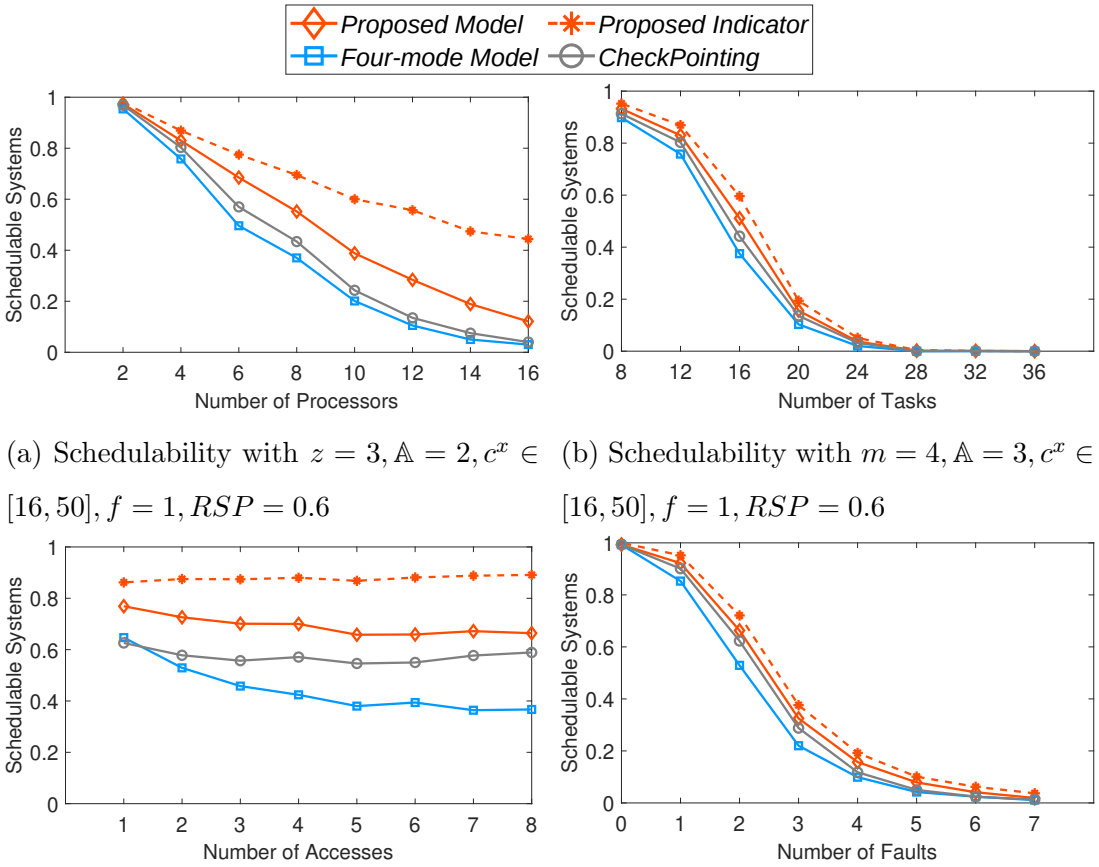
(c) Schedulability with $m = 4, z = 3, c^x \in [51, 100], f = 1, RSP = 0.8$

(d) Schedulability with $m = 4, z = 2, \mathbb{A} = 3, c^x \in [16, 50], RSP = 0.8$

Figure 20: HI mode

more overheads which is not considered in this work, the significant increase in schedulability can validate the effectiveness of the *proposed* model.

In addition, we observed that the schedulability of all models drops quickly when increasing $m \times z$ and f (i.e., Figures 19b,20b and 19d,20d). The reason is that increasing the number of tasks also increases the amount of interference and the growing number of faults imposes significant schedulability pressure. On average, for all the considered settings in Figures 19 and 20, the *proposed* model can schedule 151.5% and 25.3% more systems than the *Four-mode* model ; and



(a) Schedulability with $z = 3, \mathbb{A} = 2, c^x \in [16, 50], f = 1, RSP = 0.6$

(b) Schedulability with $m = 4, \mathbb{A} = 3, c^x \in [16, 50], f = 1, RSP = 0.6$

(c) Schedulability with $m = 4, z = 3, c^x \in [51, 100], f = 1, RSP = 0.8$

(d) Schedulability with $m = 4, z = 2, \mathbb{A} = 3, c^x \in [16, 50], RSP = 0.8$

Figure 21: Mode switch

127.8% and 11.7% more systems than the *Checkpointing* model respectively.

Figure 21 presents the resulting schedulability during a mode switch. As the same with the above figures, the *proposed* model has schedulability higher than the *Four-mode* model under all settings. In particular, with $\mathbb{A} = 8$ in Figure 21c and $f = 5$ in Figure 21d, the *proposed* model improves the number of schedulable systems by 81% and 88.1% respectively when compared to the *Four-mode* model. Lastly, for all tested system configurations, the *proposed* model outperforms the *Four-mode* and *Checkpointing* models by 65.4% and 32.5% on average, in terms

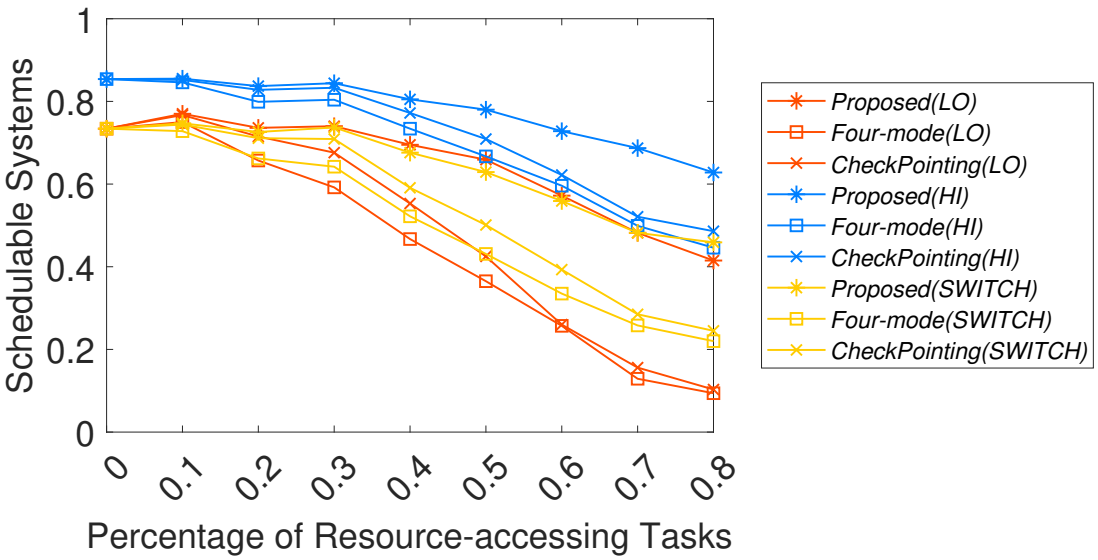


Figure 22: Schedulability with $m = 6, z = 3, \mathbb{A} = 3, c^x \in [51, 100]$

of the number of schedulable systems during a mode switch, respectively.

Figure 22 shows the schedulability of tasks in the *proposed*, *Four-mode* and *Checkpointing* models in different modes (LO, HI and SWITCH) with the variation of RSP. For systems with no shared resources (i.e., $RSP = 0$), their schedulability behaviors in all modes are the same. With the growing of RSP, the schedulability of all models decreases and the effectiveness of the *proposed* model is increasingly more obvious. This is because the increasing number of resource-accessing tasks exacerbates the resource contention and imposes pressure on the schedulability of systems for all models. Meanwhile, the increasing number of resource-accessing tasks also leads to more helpers (i.e. spinning tasks) for the head task in the FIFO queue, which are utilimed by the *proposed* model to speed up the resource-accessing routine. Therefore, the *proposed* model greatly shortens the amount of time spent tolerating faults in critical sections and reduces blocking.

3.4 Summary

This chapter introduces a fault-tolerance solution for multiprocessor MCS with shared resources. Initially, a system execution model is formulated that is fully compatible with the generic MCS having more than two criticality levels. This model can differentiate between faults that arise during normal sections and those that occur in critical sections, enabling the application of distinct fault-tolerant methods. Subsequently, a novel fault-tolerance resource-sharing protocol is devised to minimize the blocking time while ensuring reliable resource sharing. A schedulability analysis is also presented to establish timing bounds for the proposed solution. Experimental results indicate that the proposed method surpasses the state-of-the-art in terms of schedulability, showing an average improvement of up to 151.5%. This chapter represents the pioneering step of the HPRTS framework, addressing Limitations 1, 2, and 3 as outlined in Section 2.3.4 and established the success criteria SC-1 and SC-2 for the thesis.

Chapter 4

Contention-Aware Task Allocation

Real-time systems are increasingly incorporating complex functionalities, from autonomous vehicles to 5G telecommunications base stations. These systems often include tasks that need to access shared resources such as code segments, memory blocks, and I/O ports. To prevent race conditions during computations on these resources, spin locks, as recommended by the AUTOSAR standard [40], are extensively utilized. Resource sharing protocols manage access to these shared resources [15]. However, enforcing spin locks can result in resource contention due to competition for these shared resources. This problem intensifies when tasks across different processors request the same resource, leading to a busy-wait state on their host processors until the resource is granted, potentially causing significant blocking and impacting system schedulability [98].

The issue of resource contention among processors has been addressed by focusing on resource-aware task allocation within fully partitioned multiprocessor systems. These systems localize shared resources to minimize contention. Various resource-aware task allocation methods have been proposed for different scheduling methodologies and resource-sharing protocols, including FPS and the MSRP [42]. Broadly speaking, these methods fall into two categories: those that execute critical sections on the task's host processor, and those that execute on dedicated processors. The former approach, which reduces contention by assigning tasks that share resources to a single processor, may prove ineffective for complex systems [49, 62, 97]. The latter approach circumvents these issues by designating specific processors for resource access, but this introduces additional migration overhead and is suboptimal for scenarios requiring frequent resource

access [109].

This chapter presents the second step of the proposed HPRTS framework of the thesis, addressing the task allocation problem in the presence of shared resources. Initially, we introduce a Resource Contention Model (RCM) for tasks operating in systems that utilize FIFO-spin-based resource-sharing protocols to manage shared resources. This is done without restricting priority assignments or resource-sharing protocols. With the RCM, we can determine the contention intensity between tasks. Building on this foundation, we propose a contention-aware allocation algorithm designed to reduce contention between tasks when they request shared resources. We first demonstrate the evaluation of the proposed methods in homogeneous architectures, where all cores in the system are identical. Subsequently, we transition to heterogeneous architectures, where cores can operate at different frequencies at runtime, to evaluate the robustness of the proposed approaches.

This chapter unfolds as follows: Section 4.1 introduces the RCM. The contention-aware task allocation algorithm is elucidated in Section 4.2. Our evaluation results are highlighted in Section 4.3, while Section 4.4 wraps up the chapter and outlines the success criteria built for the thesis

4.1 Resource Contention Model

In this section, we introduce a novel RCM designed to approximately capture the contention intensity between tasks. The RCM considers the characteristics of tasks scheduled under a FIFO-spin-based resource-sharing protocol using preemptive FPS. Importantly, it remains decoupled from priority assignment algorithms, resource-sharing protocols, and schedulability tests. It is noteworthy that the RCM neither provides safe timing bounds for tasks with shared resources nor does it offer an exact quantification of the resource contention between tasks.

Instead, the primary objective of the RCM is to generate Contention Factor (CF) approximations with a sufficient level of accuracy among tasks. This can provide effective guidance for task allocation in order to minimize contention.

In a system with shared resources, when a task τ_i is allocated to a processor P_x, it implies that other tasks on P_x will not experience global contention (direct spin delay) from τ_i. Conversely, tasks on different processors might incur direct spin delay from τ_i when they request the same global shared resources. The RCM is developed to identify potential global contention between two tasks or two task groups, assuming they are allocated to separate processors.

As described in [98], under FIFO spin locks, each global resource is associated with a FIFO queue that contains requests from all processors based on their arriving order, in which the task at the head of the queue is always granted with the resource [104]. This leads to Lemma 5.

Lemma 5. *With FIFO spin locks, a request of τ_i to r^k can be blocked at most once by requests to r^k from a remote processor.*

Proof. For non-preemptive spin locks, a task τ_i remains non-preemptive when requesting for a shared resources until it releases the lock hence at most one request can be issued from a processor at any time instance. An example is illustrated in Example 4 in Section 2.2.4 and this is also proved in [98, 109]. As for preemptive ones, a higher-priority task τ_j can preempt τ_i when it is requesting for a resource r^k, however, τ_j cannot request r^k after preempting τ_i. Otherwise, deadlocks can occur in which τ_i cannot be served by the lock as it is preempted by a τ_j that is waiting (spinning) behind it in the FIFO queue. Thus, a processor contains at most one request to a global resource at any time instant with FIFO spin locks. □

According to Lemma 5, the contention that τ_i can incur from a task group $\mathcal{G}_a$ under FIFO spin locks is estimated by Equation 38, denoted as $\phi(\tau_i, \mathcal{G}_a)$.

The notation $N_i^k(t)$ represents the number of requests from τ_i to r^k over a time duration t, i.e., $N_i^k(t) = \left\lceil \frac{t}{T_i} \right\rceil \times N_i^k$. Assuming that τ_i and $\mathcal{G}_a$ are allocated on two different processors, τ_i can send $N_i^k(T_i)$ requests to r^k during its period T_i. Concurrently, during the same period, the maximum number of requests from tasks in $\mathcal{G}_a$ to r^k can be represented as $\sum_{\tau_j \in \mathcal{G}_a} N_j^k(T_i)$. Therefore, the *min* function is used to calculate the number of global blocking instances that τ_i can incur from $\mathcal{G}_a$ for r^k, same idea has been explained when introducing the advanced RTA for MSRP in Section 2.2.4. By accounting all the resources shared between τ_i and $\mathcal{G}_a$, the maximum amount of blocking that τ_i can incur from $\mathcal{G}_a$ is captured by $\phi(\tau_i, \mathcal{G}_a)$.

$$\phi(\tau_i, \mathcal{G}_a) = \sum_{r^k \in \mathbb{R}} \min \left\{ N_i^k(T_i), \sum_{\tau_j \in \mathcal{G}_a} N_j^k(T_i) \right\} \times c^k \tag{38}$$

By employing Equation 38, the CF approximation between $\mathcal{G}_a$ and $\mathcal{G}_b$ under FIFO-spin locks can be determined as $\Delta(\mathcal{G}_a, \mathcal{G}_b)$, as shown in Equation 39. This equation computes the summation of contention that tasks in both groups might incur if they were allocated to different processors. The term $\sum_{\tau_i \in \mathcal{G}_a} \phi(\tau_i, \mathcal{G}_b)$ calculates the sum of the CF between each task τ_i in $\mathcal{G}_a$ and the task group $\mathcal{G}_b$. Similarly, $\sum_{\tau_j \in \mathcal{G}_b} \phi(\tau_j, \mathcal{G}_a)$ computes the sum of the CF between each task τ_j in $\mathcal{G}_b$ and the task group $\mathcal{G}_a$. A higher value of $\Delta(\mathcal{G}_a, \mathcal{G}_b)$ suggests that it would be more beneficial to allocate both groups $\mathcal{G}_a$ and $\mathcal{G}_b$ to the same processor, if feasible. Notation for RCM are summarized in Table 9.

$$\Delta(\mathcal{G}_a, \mathcal{G}_b) = \sum_{\tau_i \in \mathcal{G}_a} \phi(\tau_i, \mathcal{G}_b) + \sum_{\tau_j \in \mathcal{G}_b} \phi(\tau_j, \mathcal{G}_a) \tag{39}$$

Table 9: Table of Notations for RCM

Notation	Description
r^k	A shared resource with index k
$\mathcal{G}_a$	A task group with index a
$N_i^k(t)$	Number of requests from τ_i to r^k over a time duration t
T_i	Period of task τ_i
c^k	the length of critical section of r^k
$\phi(\tau_i, \mathcal{G}_a)$	CF approximation between τ_i and a task group $\mathcal{G}_a$
$\Delta(\mathcal{G}_a, \mathcal{G}_b)$	CF approximation between two task groups $\mathcal{G}_a$ and $\mathcal{G}_b$

4.2 Contention-Aware Task Allocation

This section presents a resource-aware task allocation method for fully-partitioned systems with preemptive FPS applying FIFO-spin locks, in which shared resources are accessed by tasks directly on their host processors. Different from existing methods, the proposed method utilizes the RCM proposed in Section 4.1 that can approximate CF between tasks or task groups. Using the approximated CF as guidance, the method produces allocation decisions following the fundamental principle that always (i) group tasks with the highest CF approximation and (ii) allocate task groups with the highest approximated CF to one processor.

As the RCM does not aim to compute a safe bound on the worst-case response or blocking time, it can produce quick CF approximations with a sufficient level of accuracy without detailed knowledge of the underlying system. This provides the key to the generality of the proposed allocation method which is universally applicable in all fully-partitioned systems scheduled by preemptive FPS and with shared resources managed by FIFO-spin locks.

4.2.1 Task Grouping Based on Resource Contention

For a given set of tasks, the proposed allocation method forms a set of task groups G in which each group contains tasks with high resource contention. A utilization bound $\overline{U}$ is applied for all task groups to bound the maximum utilization allowed for one group, i.e., $\overline{U} \geq U_{\mathcal{G}_a}, \forall \mathcal{G}_a \in G$. In this work, $\overline{U}$ is set to the average utilization of the system in each processor, i.e., $\overline{U} = U_\Gamma/m$, m denotes the total number of cores in the system.

The proposed task grouping technique follows the principle of always merging two groups that have the highest CF approximation with $\overline{U}$ enforced. By doing so, only the tasks with a high resource contention will be merged into one group, even if they share certain resources with other tasks in the system. In addition, the use of $\overline{U}$ effectively avoids the creation of heavy groups that cannot be fitted into any processor. This highlights the key difference with the SPA, which tends to form large task bundles due to Limitation 4 discussed in Section 2.4.2.

Algorithm 1 presents the complete task grouping process based on the RCM. The algorithm takes all tasks in the system Γ as the input and initializes $|\Gamma|$ task groups where each group contains a single task (lines 1-4). For any two groups $\mathcal{G}_a$ and $\mathcal{G}_b$, the CF approximation is computed using the RCM, i.e., the $\Delta(\mathcal{G}_a, \mathcal{G}_b)$ in line 5. Then, the group formation starts by identifying the pair of groups with the highest CF approximation, among any two groups that can satisfy $U_{\mathcal{G}_a} + U_{\mathcal{G}_b} \leq \overline{U}$ (line 7). If such a group pair is obtained with a positive CF approximation, a new group is created by merging the two groups (line 11). In addition, the CF approximation between the newly-created group and the existing ones in G is computed for further group formation in the following iterations (line 12). This process finishes until each group is independent from others (i.e., $\Delta(\mathcal{G}_a, \mathcal{G}_b) = 0$) or no groups can be merged with $\overline{U}$ enforced (lines 8-10). Due to the former condition, groups with a single task that does not share any resources will not

Algorithm 1: Task group formation

$G = \varnothing$;
for each $\tau_i \in \Gamma$ **do**
    $\mathcal{G}_a = \{\tau_i\}$; $G = G \cup \mathcal{G}_a$;
end
compute $\Delta(\mathcal{G}_a, \mathcal{G}_b), \forall \mathcal{G}_a, \mathcal{G}_b \in G \wedge a \neq b$;
while true **do**
    $(\mathcal{G}_a, \mathcal{G}_b) = \underset{a,b}{argmax}\{\Delta(\mathcal{G}_a, \mathcal{G}_b) \mid U_{\mathcal{G}_a} + U_{\mathcal{G}_b} \leq \overline{U}\}$;
    if $\Delta(\mathcal{G}_a, \mathcal{G}_b) = 0 \vee (\mathcal{G}_a, \mathcal{G}_b) = \varnothing$ **then**
        return G;
    end
    $\mathcal{G}_a = \mathcal{G}_a \cup \mathcal{G}_b$; $G = G \setminus \mathcal{G}_b$;
    Compute $\Delta(\mathcal{G}_a, \mathcal{G}_s), \forall \mathcal{G}_s \in G \wedge a \neq s$;
end
return G;

be merged during the entire process. Example 9 illustrates the group formation process using four tasks and Characteristic 1 summarizes the key that addresses Limitation 4 discussed in Section 2.4.2.

Example 9. *For a task set* $\{\tau_1, ..., \tau_4\}$*, they are initialized as a set of groups* $\{\mathcal{G}_1 = \{\tau_1\}, ..., \mathcal{G}_4 = \{\tau_4\}\}$*, in which each group contains one task. Assuming* $\Delta(\mathcal{G}_1, \mathcal{G}_4)$ *is the highest among all groups, they will be merged together if* $\overline{U}$ *is satisfied, i.e.,* $\{\mathcal{G}_1 = \{\tau_1, \tau_4\}, ..., \mathcal{G}_3 = \{\tau_3\}\}$*. Then, if* $\Delta(\mathcal{G}_2, \mathcal{G}_3)$ *is the highest,* $\mathcal{G}_2$ *and* $\mathcal{G}_3$ *are merged even if they also have a positive CF approximation with* $\mathcal{G}_1$*, i.e.,* $\{\mathcal{G}_1 = \{\tau_1, \tau_4\}, \mathcal{G}_2 = \{\tau_2, \tau_3\}\}$*. However, under SPA all tasks are grouped together regardless of the resource contention.*

Characteristic 1. *The proposed method forms a set of task groups in which tasks in each group has the highest contention. In addition,* $\overline{U}$ *is applied to avoid forming large task groups that cannot be allocated in a single processor. This*

addresses Limitation 4.

4.2.2 Allocation of Task Groups on Processors

Algorithm 2: Resource-aware task allocation

```
1  for each G_a ∈ G do
2  |  Ω_a = Σ_{τ_i ∈ G_a} Δ({τ_i}, G_a \ τ_i);
3  end
4  for each x ∈ [1... min{|G|, m}] do
5  |  G_a = argmax_a {Ω_a | ∀G_a ∈ G};
6  |  P(τ_i) = P_x, ∀τ_i ∈ G_a;    G = G \ G_a;
7  end
8  while G ≠ ∅ do
9  |  P_x = argmin_x {U_{Γ(P_x)} | ∀P_x ∈ Λ};
10 |  G_a = argmax_a {Δ(G_a, Γ(P_x)) | ∀G_a ∈ G};
11 |  if U_{P_x} + U_{G_a} ≤ 1 then
12 |  |  P(τ_i) = P_x, ∀τ_i ∈ G_a;    G = G \ G_a;
13 |  else
14 |  |  for each τ_i ∈ G_a, highest Δ({τ_i}, Γ(P_x)) first do
15 |  |  |  if U_{P_x} + U_{τ_i} ≤ 1 then
16 |  |  |  |  P(τ_i) = P_x;    G_a = G_a \ τ_i;
17 |  |  |  end
18 |  |  end
19 |  |  if no task in G_a is allocated to P_x then
20 |  |  |  return unfeasible;
21 |  |  end
22 |  end
23 end
24 return {P(τ_i), ∀τ_i ∈ Γ};
```

With G constructed, Algorithm 2 is performed that maps G to a set of processors Λ. The algorithm starts by computing a weight Ω_a for each $\mathcal{G}_a$ based

on the CF. This factor reflects the degree of the internal resource contention between tasks in $\mathcal{G}_a$, which is calculated by summing the CF between each $\tau_i \in \mathcal{G}_a$ and other tasks in the group $\mathcal{G}_a \backslash \tau_i$ (line 1-3).

The allocation is then performed in two steps. The first step produces an initial allocation for up to m groups in G based on Ω_a (lines 4-7). Then, the second step decides the allocations of the rest of the task groups based on the guidance of the RCM (lines 8-23).

In the first step, the method identifies at most $\min\{|G|, m\}$ task groups with the highest Ω_a and allocates each of these groups to one processor (lines 5-6). For groups that have the same Ω_a, the one with a higher utilization is taken. This is feasible as all groups are formed with $\overline{U}$ enforced and $\overline{U} \leq 1$. By localizing tasks in these groups, we mitigate the contention between the tasks with the highest CF approximations in the system.

Following the initial allocation, the second step maps the unallocated groups following the rule of *the P_x with the least $U_{\Gamma(P_x)}$ first and the $\mathcal{G}_a$ with the highest $\Delta(\mathcal{G}_a, \Gamma(P_x))$ first* (lines 9-10). $U_{\Gamma(P_x)}$ return the total utilization of tasks on a processor with index x and $\Delta(\mathcal{G}_a, \Gamma(P_x))$ return the CF between a task group $\mathcal{G}_a$ and $\Gamma(P_x)$. This approach selects processor with least utilization and pick the group in G that has the highest CF with task on the selected processor. This increases the chance of fitting a complete task group to the processor with least utilization and can reduce resource contention in a general case.

If $\mathcal{G}_a$ can be fitted into P_x, all tasks in the group are assigned to this processor (lines 11-12). Otherwise, to increase the success ratio of the allocation method with reduced contention, tasks in $\mathcal{G}_a$ are allocated individually, in which the one with the highest $\Delta(\{\tau_i\}, \Gamma(P_x))$ is always assigned to P_x. However, under this case, if no task in G_a can be assigned to P_x (i.e., the processor with the lowest utilization), the algorithm returns immediately with no feasible allocation being

found (lines 19-21). The complete process finishes until every task is assigned with an allocation, i.e., $P(\tau_i), \forall \tau_i \in \Gamma$. With the above descriptions, we conclude the following two characteristics of the proposed method. The notations related to the allocation algorithm is summarized in Table 10.

Characteristic 2. *The proposed method allocates tasks to processors based on the RCM without the need for utilizing a specific schedulability test iteratively, and hence, addresses Limitation 5.*

Characteristic 3. *Tasks under the proposed method do not require additional runtime facilities, e.g., migrations in [100], and have negligible runtime overhead that is equivalent to the traditional fully-partitioned system. This addresses Limitation 6.*

This concludes the description of the proposed task allocation method. The time complexity of the proposed method is $O(n^2)$, as at most $||\Gamma| \times |\Gamma||$ iterations are required to (i) compute $\Delta(\mathcal{G}_a, \mathcal{G}_b)$ for all group pairs in G, (ii) form task groups, and (iii) allocate the task groups to processors.

Table 10: Table of Notations for Task Allocation Algorithm

Notation	**Description**
Γ	The set of tasks in the system
$\Gamma(P_x)$	Set of tasks on processor P_x
G	Set of task groups created
Λ	Set of processors
Ω_a	Weight for each task group $\mathcal{G}_a$ based on the internal CF
$\Delta(\mathcal{G}_a, \Gamma(P_x))$	Contention Factor between a task group $\mathcal{G}_a$ and tasks on processor P_x
$U_{\Gamma(P_x)}$	Total utilization of tasks on a processor P_x

4.3 Evaluation

This section evaluates the performance of the proposed resource-aware task allocation method against the state-of-the-art methods in terms of both effectiveness and efficiency. The following competing methods are considered: (i) a combination of Worst-Fit, Best-Fit, First-Fit, and Next-Fit heuristics (see Section 2.4.1) as the baseline [97] (denoted as *Any-Fit*); (ii) the *SPA* in [62]; (iii) the *ROP* with the non-preemptive protocol applied [57]; and (iv) the proposed Resource-aware task Allocation with the RCM for FIFO spin locks (denoted as *OUR*).

4.3.1 Experimental Setup

Our evaluations, grounded in synthetic but realistic values, employ a similar experimental setup as used in [98, 109]. Our synthetic tests do not rely on extraneous variables such as caching, networking, communications, or distance to memory. We evaluate systems with m cores, which are selected from a range $m \in [4, 32]$. z is denoted as the number of tasks per core, and it varies in the range $z \in [1, 9]$. As the number of cores increases, new tasks will also be added to the system. Hence, the total number of tasks in the system is calculated as $m \times z$. The utilization of tasks is computed using the UUniFast-Discard algorithm [35], given a total utilization of $0.1 \times m \times z$.

Task periods T_i are randomly generated from the range [1 ms, 1000 ms] in a log-uniform distribution, with implicit deadline $D_i = T_i$. The Rate Monotonic Priority Ordering is applied to generate task priorities. The total computation time (including the time spent executing with shared resources) is then obtained and denoted as $\widehat{C_i} = U_{\tau_i} \times T_i$.

Each core is set to be associated with a shared resource, hence the number of shared resources is also equal to m. Each resource has a length randomly decided within a given range $c^k \in [1\mu s, 100\mu s]$. A resource sharing factor $RSP = 0.3$ is

applied to define that 30% of tasks in the system will request shared resources. Each of these tasks can randomly request $[1, m]$ resources. The maximum number of accesses from a task to a resource is chosen from $\mathbb{A} = [1, 40]$. If $\mathbb{A} = 10$, then the task randomly accesses the resource from 1 to 10 times. Let $C_i^{\mathbb{R}}$ denote the total resource computation time of τ_i. We enforce that $\widehat{C_i} - C_i^{\mathbb{R}} \geq 0$, with $C_{\tau_i} = \widehat{C_i} - C_i^{\mathbb{R}}$. For each system configuration, 1000 systems are generated and tested by each competing method.

In our extended experimental framework for a heterogeneous architecture, we work with a system of m cores. These cores exhibit a range of computational speeds, evenly distributed from 0.5 to 1. Specifically, the first core, being the fastest, operates at a speed factor of 0.5. This means tasks on this core execute at $0.5 \times C_{\tau_i}$ of their computation time. If a task access a resource r^k on this core, it will operate at $0.5 \times c^k$ of the resource computation time. As we progress to subsequent cores, the speed factor increases evenly, ensuring that each core is uniformly slower than the previous. By the time we reach the m-th core, the slowest in the setup, the speed factor is 1, and tasks execute using their original computation times: C_{τ_i} for general computation and c^k when accessing r^k. This setup provides a systematic gradient of performance across the cores. In this manner, since tasks typically compute for shorter durations, we intensify the system's schedulability pressure to highlight the differences between each method. This is achieved by expanding the range to $c^k \in [1\mu s, 360\mu s]$, which in turn increases the blocking time between tasks when accessing shared resources.

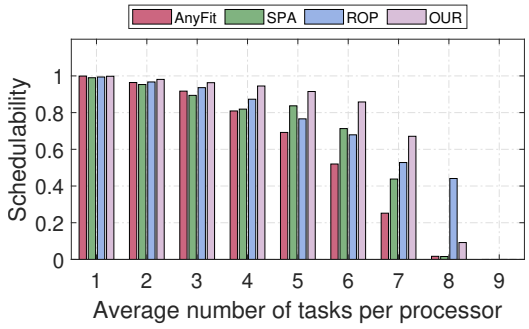
Unless specified in the allocation method (e.g., *ROP*), the MSRP is applied to manage the shared resources in the system, which is a FIFO spin-based protocol that is common-seen in multiprocessor real-time systems [57, 97, 105]. For *Any-Fit*, SPA and *OUR*, the holistic schedulability analysis for MSRP as demontrated in 2.2.4 is applied to evaluate system schedulability [98, 109]. In addition, a cost

of $8.35\mu s$ for each migration triggered by *ROP* is added in its schedulability test. This overhead is measured and reported by [104, 109] under the Linux operating system. It is worth noting that the migration cost of tasks can vary due to many factors, such as task size and hardware architectures. Here, we apply a reasonable value for our synthetic evaluation. To precisely calculate the task migration cost in Linux, tracing tools like *perf* or *ftrace* [31] are often used to measure the time from the decision to migrate a task until its resumption on a new processor, considering all associated overheads. Ensure accuracy by repeatedly measuring under various system loads and conditions, and by thoroughly understanding the impacts of specific hardware and kernel versions.

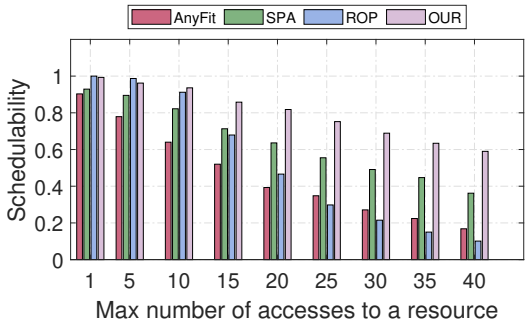
4.3.2 Performance Evaluation for Homogeneous Architecture

Figure 23 presents the schedulability performance of all methods under homogeneous architecture with varying parameters: the number of tasks per processor (z), the maximum number of accesses to a resource ($\mathbb{A}$), the number of processors (m), and the length of critical sections (c^k).

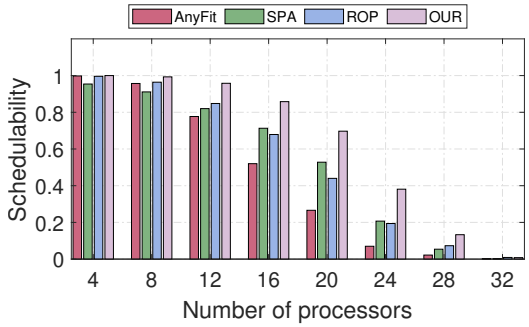
Figure 23a presents the system schedulability of the competing methods with a varied z. From the figure, we can observe that the proposed *OUR* constantly outperforms SPA, where both methods are performed based on the resource usage of the system. This validates the effectiveness of the proposed allocation and the RCM. In addition, *OUR* maintain an advantage over all other methods with $z = [1, 7]$. Specifically, the *OUR* outperforms the *ROP* by 19.5% with $z = 5$. The reason for this observation is that *ROP* have a certain degree of pessimism that undermines their effectiveness (Limitations 6 in Section 2.4.2). In contrast, the proposed allocation method addresses the limitation, i.e., has negligible runtime overhead. However, *OUR* lost its advantage with $z \geq 8$, i.e., when $\overline{U}$ is high. In this case, task groups with a high utilization can be formed



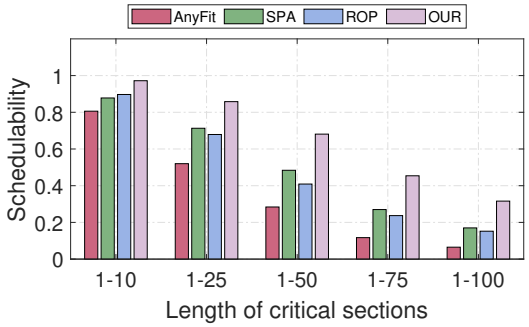
(a) Schedulability with $m = 16$, $\mathbb{A} = 15$, $c^k = [1us, 25us]$.



(b) Schedulability with $z = 6$, $m = 16$, $c^k = [1us, 25us]$.



(c) Schedulability with $z = 6$, $\mathbb{A} = 15$, $c^k = [1\mu s, 25\mu s]$.



(d) Schedulability with $z = 6$, $m = 16$, $\mathbb{A} = 15$.

Figure 23: Homogeneous schedulability figures.

which are hardly schedulable. This leads to future work of an adaptive utilization bound that further enhances the performance of the proposed methods.

Similar observations are obtained in Figures 23b and 23c with an increased $\mathbb{A}$ and m respectively, *OUR* outperforms all competing methods in most cases. It is worth noting that with an increased $\mathbb{A}$, the schedulability of *ROP* drops quickly. This is because the intensive resource accesses cause both high migration overhead and interference as all requests to a resource are accessed on a dedicated processor. In addition, the proposed methods constantly outperform *SPA* and *ROP* with an increased c^k, as shown in Figure 23d. For example, the *OUR* outperforms *ROP* by 91.6% with $c^k = [1\mu s, 75\mu s]$. This is because the increase

Table 11: Computation cost (in ms) with $\mathbb{A} = 15$, $c^k = [1us, 25us]$.

$m = 16$,	SPA		ROP		OUR	
$z =$	avg.	std.	avg.	std.	avg.	std.
2	0.2	2.6	21.6	16.4	1.9	6.3
4	0.1	0.9	72.9	34.5	8.7	9.0
6	0.1	0.1	125.4	65.5	28.1	11.2
8	0.1	0.1	238.6	259.5	59.5	16.9
$z = 6$,	SPA		ROP		OUR	
$m =$	avg.	std.	avg.	std.	avg.	std.
8	0.1	1.0	13.7	12.4	2.1	4.9
16	0.1	0.8	135.6	63.9	31.5	10.4
24	0.2	0.1	524.3	336.0	118.8	25.8
32	0.3	0.6	1391.6	725.4	371.1	61.9

in the resource length causes a higher interference on a task when accessing r^k by other requests on the same processor. This observation is consistent with that in [57].

Overall, as illustrated in Figure 23, the proposed approach consistently outperforms the state-of-the-art in terms of schedulability for the majority of cases. When evaluating all scenarios where *OUR* holds an advantage, it outperforms the search-based method with the inclusion of migration overheads (*ROP*) by an average of 74% in terms of schedulability.

Finally, we report the computation cost of the competing methods on an Intel i5-4460 processor with a CPU frequency of 3.20GHz. Table 11 presents the average (avg.) and standard deviation (std.) of the computation cost of the evaluated methods with varied z and m. As shown in the table, the *SPA* has the lowest computation cost due to its simplicity. As for the *OUR*, it requires much

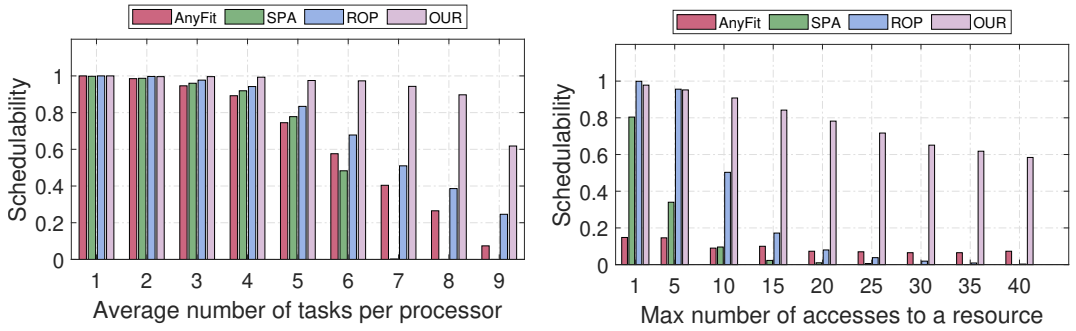
less computation time compared to both *ROP*. Specifically, the computation cost of *ROP* is about 5.9x of *OUR* on average for all settings shown in the table. In particular, although with a high schedulability, the *ROP* demonstrates a 4x the computation cost of the *OUR* on average with $z = 8$. However, in most cases, the *OUR* achieves a higher schedulability than all other methods with a much less computation cost. The reason is *ROP* use a searching method that requires iterative computations to bound the WCRT of tasks for each allocation choice, resulting in a much higher cost to find a feasible allocation solution.

In summary, the proposed allocation can outperform the state-of-the-art methods in schedulability with a much less computation cost. The results validate that the constructed RCM can provide effective guidance without details of the underlying system, and verify the effectiveness and efficiency of the proposed allocation method.

4.3.3 Performance Evaluation for Heterogeneous Architecture

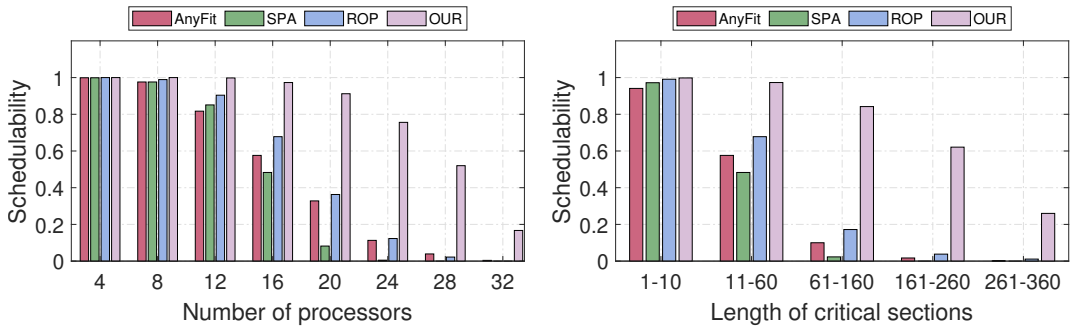
In Figure 24, we showcase the schedulability performance across various methods for a heterogeneous setting. The parameters under consideration include the tasks per processor (z), maximum resource accesses ($\mathbb{A}$), processor count (m), and critical section lengths (c^k).

The results depicted in Figure 24a reveal that for $z = 1$, all algorithms, with the exception of SPA, achieve a schedulability rate close to or equal to 100%. This suggests that the system experiences minimal schedulability pressure. As the number of tasks per processor z rises, the *OUR* method consistently outperforms other algorithms in terms of schedulability rate. Up to $z = 7$, *OUR* maintains a rate above 94%, while *ROP* declines to 51% and *SPA* registers only 0.2% schedulable systems. This indicates that the *OUR* method offers enhanced robustness and reliability under diverse conditions, outshining other algorithms



(a) Schedulability with $m = 16$, $\mathbb{A} = 15$, $c^k = [11us, 60us]$.

(b) Schedulability with $z = 6$, $m = 16$, $c^k = [61us, 160us]$.



(c) Schedulability with $z = 6$, $\mathbb{A} = 15$, $c^k = [11\mu s, 60\mu s]$.

(d) Schedulability with $z = 6$, $m = 16$, $\mathbb{A} = 15$.

Figure 24: Heterogeneous schedulability figures.

in both homogeneous and heterogeneous architectures.

On the other hand, the results presented in Figure 24b show that at $\mathbb{A} = 1$, *OUR* achieves a schedulability rate of 97.8%, which is significantly higher than *AnyFit*'s rate of 14.8%. *SPA* performs well, with a schedulability rate of 80.4%, while *ROP* have rates of 99.9%. As $\mathbb{A}$ increases to 20, as shown in Figure 24b, *OUR* continues to lead, with a schedulability rate of 78.2%. This outperforms *AnyFit* by 70.9% (equivalent to 709 more systems being schedulable), *SPA* by 77.2% (772 more systems schedulable), and *ROP* by 70.2% (702 more systems schedulable). As $\mathbb{A}$ progresses to 40, Figure 24b illustrates that despite the general decreasing trend in schedulability rates, *OUR* continues to exhibit relatively

better performance,

In both Figures 24c and 24d, we demonstrate the dominance of all algorithms as various settings increase. Specifically, *OUR* consistently stands out in schedulability, maintaining a 16.7% rate at $m = 32$. In contrast, *AnyFit* starts robustly but drops to 0.4% by $m = 32$, and *SPA* reaches 0% by $m = 28$. *ROP*, while declining, still lags notably behind *OUR*, hitting 0% at $m = 32$. Turning to Figure 24d, as c^k extends to $[11, 60]$, *OUR* retains a strong 97.3% rate, significantly outperforming both *ROP* and *SPA* by margins of 43.5% and 101%, respectively. These results underscore *OUR*'s adeptness in managing varying system pressures and resource contention.

In conclusion, the *OUR* algorithm consistently exhibits superior performance across various scenarios compared to *AnyFit*, *SPA*, and *ROP*. This dominance mirrors its performance under homogeneous architecture. The pronounced performance gap stems from the combined effects of our RCM, which precisely captures the contention level between tasks, and our allocation algorithm, which markedly reduces global resource contention by assigning contention-intensive tasks to the same core.

4.4 Summary

In this chapter, we explored the task allocation challenge within the context of shared resources, marking the second phase of our proposed HPRTS framework. We introduced the RCM, tailored for tasks in systems that employ FIFO-spin-based resource-sharing protocols, without imposing constraints on priority assignments or the protocols themselves. This model enables us to gauge the intensity of contention between tasks. Leveraging the insights from the RCM, we unveiled a contention-aware allocation algorithm aimed at minimizing contention during shared resource requests.

Our evaluations commenced with homogeneous architectures to ensure a uniform core environment. In most cases, the proposed approach outperforms state-of-the-art methods. Specifically, in scenarios where our method has an advantage, it surpasses the search-based method (ROP) by an average of 74% in terms of schedulability. Furthermore, the computational cost of ROP is approximately 5.9 times of the proposed approach on average. We subsequently extended our evaluations to the more intricate heterogeneous architectures, where core frequencies can differ. The performance disparity between our proposed method and other methods becomes even more pronounced. In many instances, while our method still maintains a significant number of schedulable systems, other methods are scarcely schedulable. Overall, the proposed approaches effectively address the Limitation 4, 5 and 6 as defined in Section 2.4.2 and established the success criteria SC-3 and SC-4 of the thesis.

Chapter 5

Precise Response Time Analysis for Multiple DAG Tasks

With emerging real-time application scenarios, such as in autonomous systems and Ultra-Reliable Low Latency Communications (URLLC) in the industrial automation domain, complex functionalities are increasingly deployed in multi-core real-time systems. In these systems, the functionalities can be delivered in a concurrent fashion, but are subject to execution dependencies at certain points of execution [96]. To reflect the complex dependency and parallelism that widely exist between computations in the system, the DAG task model has received much attention in the real-time systems community [10].

A DAG task contains a set of nodes and directed edges, in which each node indicates a code segment that must be executed in a sequential manner and a directed edge connecting two nodes indicates their execution dependency [10]. Any two nodes that have no execution dependencies between each other can be executed in parallel across multiple cores. Compared to the traditional sporadic task model [21] in which tasks have no execution dependencies between each other, the DAG task abstraction provides a more realistic model that better describes the internal execution relationships within the system. However, it also imposes new research challenges in ensuring system schedulability as the traditional bound on the WCRT of regular task models (e.g. sporadic and periodic) does not take these existing execution dependencies within DAGs into account.

This chapter constructs the last step of the proposed HPRTS, we move towards the exploration of more complicated task model. We focus on periodic DAG tasks running on a homogneous multi-core system with a global limited

preemption scheme. The principal contribution of this chapter is a novel priority-explicit RTA that advances the industrial usage of RTA for DAGs in real-world systems. The proposed analysis produces tighter upper bounds for DAG tasks by precisely bounding the intra-task and inter-task delay of a DAG task.

The structure of the following sections is as follows: Section 5.1 introduces a novel RTA for DAG tasks, Section 5.2 presents the evaluation results, and Section 5.3 concludes with some key findings.

5.1 Response Time Analysis for DAG Tasks

As described in Section 2.5.3, the existing RTA calculates the WCRT of a single DAG (or makespan) first, and then adds the inter-task delay upon the single-DAG RTA, which can result in a pessimistic bound leading to Limitations 7 and 8.

In this section, we construct a finer-grained RTA that provides the WCRT bounds for both single- and multi-DAG systems by fully exploring the potential parallelism of each node in the system to address the limitations. For clarity, Table 12 concludes the necessary notations for RTA of single-DAG systems.

As mentioned in Section 2.5.1, the proposed RTA assumes that nodes between two DAG tasks are independent, they neither have constraints nor share resources between each other. Moreover, nodes within a DAG are assigned individual priorities. If the priority of τ_i exceeds that of τ_k (i.e., $Pri(\tau_i) > Pri(\tau_k)$), then every node in τ_i will have a priority greater than those in τ_k (i.e., $Pri(v_a) > Pri(v_b)$ for $\forall v_a \in V_i$ and $v_b \in V_k$). Furthermore, the proposed RTA is applicable to any priority assignment. When referring to the term *delay* individually in this chapter, it encompasses both interference and blocking delays.

Table 12: Table of Notations for Single DAG Anlaysis

Notation	Description
$\text{Pri}(\tau_i)$	Priority of task τ_i
V_i	Set of nodes in task τ_i
Ft_{v_j}	Worst-case finish time of node v_j
St_{v_j}	Worst-case starting time of node v_j
C_{v_j}	Worst-Case Execution Time (WCET) of node v_j
I_{v_j}	Nodes that can cause an intra-task delay to node v_j
m	Total number of cores in the system
$I_{v_j}^{remove}$	Set of nodes that will never delay the execution of v_j
$I_{v_j}^{potential}$	Set of nodes that can potentially cause intra-task delay to a node v_j
$I_{v_j}^{hi}$	High-priority nodes in $I_{v_j}^{potential}$
$I_{v_j}^{lo}$	Low-priority nodes in $I_{v_j}^{potential}$
$I_{v_j}^{lomax}$	$m-1$ largest nodes among the low-priority nodes in $I_{v_j}^{potential}$
$I_{v_j}^{lopre}$	Ancestor nodes of $I_{v_j}^{lomax}$ among the low-priority nodes in $I_{v_j}^{potential}$

5.1.1 RTA for Single-DAG Systems

To analyze the WCRT of a DAG task, we aim to identify the worst-case finish time of each node and then identify the node that finishes the latest in the DAG task. Equation 40 presents the worst-case makespan of a single DAG task, denoted as R_i. Notation Ft_{v_j} defines the worst-case finish time of node v_j and $\max_{v_j \in V_i}\{Ft_{v_j}\}$ calculates the node that has the largest finishing time in the DAG.

$$R_i = \max_{v_j \in V_i}\{Ft_{v_j}\} \tag{40}$$

Then for a given node v_j, its worst-case finish time is computed as the sum

of its worst-case starting time St_{v_j} and its WCET C_{v_j}, as shown in Equation 41.

$$Ft_{v_j} = St_{v_j} + C_{v_j} \tag{41}$$

For a node v_j, it is allowed to start its execution if all of its predecessors have finished their executions. In addition, when v_j is eligible to execute, it can incur the intra-task delay imposed from the concurrent nodes with a higher priority in the DAG task. Equation 42 bounds the worst-case starting time of v_j, in which I_{v_j} gives the nodes that can cause an intra-task delay on node v_j. This equation iterates through each predecessor node $v_k \in pre(v_j)$, and computes the worst-case finish time of v_k and the associated intra-task delay that v_j can incur, and hence, derives a safe bound [53].

$$St_{v_j} = \max_{v_k \in pre(v_j)} \left\{ Ft_{v_k} + \left\lceil \frac{C_{I_{v_j} \setminus I_{v_k}}}{m} \right\rceil \right\} \tag{42}$$

In addition, for the computation of the intra-task delay, we exclude the delay (i.e. $I_{v_j} \setminus I_{v_k}$) that has been accounted for in Ft_{v_k} to avoid repetitive calculations. The term $C_{I_{v_j} \setminus I_{v_k}}$ represents the total workload that could delay v_j, and the final intra-task delay of v_j is given by $\lceil \frac{C_{I_{v_j} \setminus I_{v_k}}}{m} \rceil$. Below we present details on the identification of nodes in I_{v_j}.

Under the limited preemptive scheduling scheme with a defined priority assignment policy, a node v_j will not necessarily be delayed by all of its concurrent nodes $S(v_j)$ (defined in Definition 2). To bound the amount of delay suffered by v_j from $S(v_j)$, we first identify the nodes that can never delay v_j.

Lemma 6. *For a pair of concurrent nodes v_k and v_j with $Pri(v_k) < Pri(v_j)$, if v_k is eligible to execute (i.e., all of its predecessors have finish executing) at the same time as or later than v_j, v_k and its descendants will not impose a delay to v_j.*

Proof. If v_k is eligible to execute at the same time as or later than v_j, because $Pri(v_k) < Pri(v_j)$, once a core is available the scheduler will select v_j to execute first. Therefore, v_k and its descendants ($des(v_k)$) will not delay v_j. □

Lemma 7. *For a node v_j, nodes in the same DAG that are eligible to execute at the same time with or later than v_j are computed by $\eta(v_j)$, as shown in Equation 43.*

$$\eta(v_j) = \left\{ v_k | v_k \in V_i \wedge pre(v_j) \subseteq pre(v_k) \wedge v_j \neq v_k \right\} \tag{43}$$

Proof. For a node $v_k \in V_i$ and $v_j \neq v_k$, the predecessors of v_k ($pre(v_k)$) containing all the predecessors of v_j ($pre(v_j)$) can be expressed as $pre(v_j) \subseteq pre(v_k)$. Hence, the predecessors of v_k can finish later than or at the same time as the predecessors of v_j. Finally, we can deduce that v_k is eligible to execute at the same time with or later than v_j □

Based on Lemma 6 and 7, the set of nodes that cannot impose any delay to the node v_j (denoted as $I_{v_j}^{remove}$) can be identified in Equation 44. This equation iterates over all the nodes in the DAG and appends v_k and its descendants ($des(v_k)$) in $I_{v_j}^{remove}$, if the following conditions are satisfied: 1) the priority of v_k is smaller than v_j ($Pri(v_k) < Pri(v_j)$); and 2) $v_k \in \eta(v_j)$ or any of its ancestors belongs to $\eta(v_j)$ (i.e., $anc(v_k) \cap \eta(v_j) \neq \emptyset$).

$$I_{v_j}^{remove} = \bigcup_{v_k \in V_i} \begin{cases} v_k \cup des(v_k), & if\ Pri(v_k) < Pri(v_j) \wedge \\ & (v_k \in \eta(v_j) \vee anc(v_k) \cap \eta(v_j) \neq \emptyset) \\ \emptyset, & \text{otherwise.} \end{cases} \tag{44}$$

Theorem 2. *Equation 44 bounds the node set that will never delay the execution of v_j.*

Proof. For a node v_k with priority $Pri(v_k) < Pri(v_j)$ that satisfies $v_k \in \eta(v_j)$, it cannot impose any delay to node v_j according to Lemma 6 and 7. Moreover,

if any ancestors of v_k belong to $\eta(v_j)$ (i.e. $anc(v_k) \cap \eta(v_j) \neq \emptyset$), the ancestor is eligible to execute at the same time as or later than v_j according to Lemma 7. Accordingly, v_k is eligible to execute at the same time as or later than v_j, hence, v_k cannot impose any delay to node v_j according to Lemma 6. □

Based on Theorem 2, Equation 45 bounds the set of nodes that can potentially cause intra-task delay with a node v_j, denoted as $I_{v_j}^{potential}$. As shown in the equation, $I_{v_j}^{potential}$ is computed by excluding the nodes that cannot delay v_j from the concurrent node of v_j (i.e. $S(v_j) \setminus I_{v_j}^{remove}$).

$$I_{v_j}^{potential} = S(v_j) \setminus I_{v_j}^{remove} \tag{45}$$

Lemma 8. *For a node v_j, if the nodes in $I_{v_j}^{potential}$ cannot occupy all m cores in the system, then v_j will not suffer from the intra-task delay.*

Proof. Assuming all nodes in $I_{v_j}^{potential}$ execute on $m-1$ cores of a system, when v_j arrives, it will directly execute on the idle core without being delayed. □

Example 10. *As shown in Figure 25, the set of nodes that can potentially delay v_j is $I_{v_j}^{potential} = \{v_1, v_2, v_3, v_4\}$. Due the dependencies of nodes in $I_{v_j}^{potential}$, they can occupy at most two cores, therefore v_j can execute on the core available directly without incurring any delay.*

Let $path(I_{v_j}^{potential})$ denote a function that computes the largest number of cores that $I_{v_j}^{potential}$ can occupy. Based on Lemma 8, I_{v_j} can be computed by Equation 46. As shown in this equation, when $path(I_{v_j}^{potential}) < m$ (i.e., the potential delaying nodes cannot take all m cores), v_j does not incur any intra-task delay, and hence, I_{v_j} can be directly set to empty. This approach takes into account scenarios where nodes can execute in parallel, thereby avoiding unnecessary delay calculations between nodes. This addresses Limitation 7, as illustrated in Section 2.5.3.

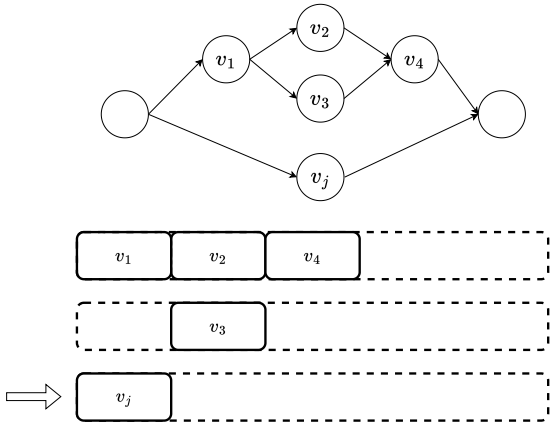


Figure 25: An example of the interference-free execution.

Otherwise, if $path(I_{v_j}^{potential}) \geq m$, v_j can incur a certain amount of intra-task delay from $I_{v_j}^{potential}$. As shown in Equation 46, under this case, the nodes that cause this delay can be categorized into the following three groups.

1. $I_{v_j}^{hi}$: the high-priority nodes in $I_{v_j}^{potential}$,
2. $I_{v_j}^{lomax}$: the $m-1$ largest nodes among the low-priority nodes in $I_{v_j}^{potential}$, and
3. $I_{v_j}^{lopre}$: the ancestor nodes of $I_{v_j}^{lomax}$ among the low-priority nodes in $I_{v_j}^{potential}$.

$$I_{v_j} = \begin{cases} \emptyset, & if\ path(I_{v_j}^{potential}) < m \\ I_{v_j}^{hi} \cup I_{v_j}^{lomax} \cup I_{v_j}^{lopre}, & \text{otherwise} \end{cases} \tag{46}$$

Equation 47 provides the computation for $I_{v_j}^{hi}$, which takes all the high-priority nodes in $I_{v_j}^{potential}$ into account. In the worst case, when v_j arrives, all nodes in $I_{v_j}^{hi}$ have just started executing, therefore, $I_{v_j}^{hi}$ is included in I_{v_j}.

$$I_{v_j}^{hi} = \left\{ v_k | v_k \in I_{v_j}^{potential} \wedge Pri(v_k) > Pri(v_j) \right\} \tag{47}$$

Under the global limited preemption scheme, in a multi-DAG scenario, the source node of DAG may experience blocking from up to m low-priority nodes.

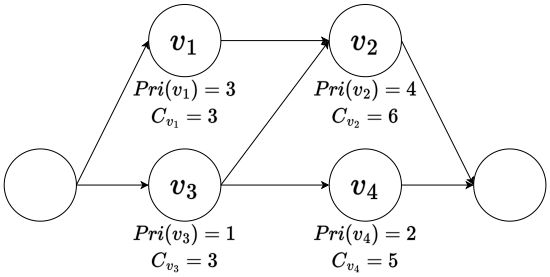


Figure 26: An example of a DAG task.

This assumes that when the source node arrives, all the cores are occupied by low-priority nodes from other DAG tasks. However, in a single DAG scenario, the source node will be the starting node, therefore, incur no blocking delay [86, 92].

In both scenarios the remaining nodes may experience blocking from up to $m-1$ low-priority nodes. This is because before the node arrives, one core will be taken up by its predecessor and the rest $m-1$ cores can be taken by the concurrent low-priority nodes. Proofs of the above statements on the low-priority intra-task blocking delay can be found in Lemmas 4.2 and 4.3 in [92]. Therefore, the low-priority nodes in $I_{v_j}^{potential}$ that can cause a delay to v_j can be computed by Equations 48 and 49, in which $I_{v_j}^{lo}$ returns all the low-priority nodes in $I_{v_j}^{potential}$ and $I_{v_j}^{lomax}$ identifies the $m-1$ nodes in $I_{v_j}^{lo}$ that can cause the maximum blocking delay. Example 11 demonstrates an example of bounding $I_{v_j}^{lomax}$.

$$I_{v_j}^{lo} = \left\{ v_k | v_k \in I_{v_j}^{potential} \wedge Pri(v_k) < Pri(v_j) \right\} \tag{48}$$

$$I_{v_j}^{lomax} = \overset{m-1}{\max}\{I_{v_j}^{lo}\} \tag{49}$$

Example 11. *As shown in the Figure 26, the concurrent nodes of* v_1 *is* $S(v_1) = \{v_3, v_4\}$*. As the priority of* v_1 *(*$Pri(v_1) = 3$*) is higher than the priorities of* v_3 *(*$Pri(v_3) = 1$*) and* v_4 *(*$Pri(v_4) = 2$*), we have* $I_{v_1}^{lo} = \{v_3, v_4\}$*. For a system with two cores, the maximum number of low-priority tasks that node* v_1 *can execute*

concurrently with is 1 and given that $C_{v_3} < C_{v_4}$, the intra-task blocking of low-priority nodes is $I_{v_1}^{lomax} = \{v_4\}$ in this example.

According to Figure 26, $S(v_1) = \{v_3, v_4\}$ and $I_{v_1}^{lomax} = \{v_4\}$, $S(v_2) = \{v_4\}$ and $I_{v_2}^{lomax} = \{v_4\}$. In the present instance, v_4 will only be counted as interference of v_1 and not for v_2, since v_1 is the predecessor of v_2 and Equation 42 excludes repeated interference from being counted more than once. However, there may exist a situation that v_3 delays v_1 and v_4 delays v_2, as this situation still meets the constraint that each node within a DAG can suffer delay from at most $m-1$ low-priority node mentioned above. We can assume v_3 will impose blocking delay to v_1 as well to derive a safe upper bound for the intra-task delay. Such nodes are denoted as $I_{v_j}^{lopre}$ and are bounded in Equation 50. The $I_{v_j}^{lopre}$ iterates through nodes in $I_{v_j}^{lomax}$ denoted as v_p, and identifies each node that belongs to $I_{v_j}^{lo}$ but is not in $I_{v_j}^{lomax}$ (i.e., $v_k \in (I_{v_j}^{lo} \setminus I_{v_j}^{lomax})$), and is an ancestor of v_p (i.e., $v_k \in anc(v_p)$).

$$I_{v_j}^{lopre} = \bigcup_{v_p \in I_{v_j}^{lomax}} \{v_k | v_k \in (I_{v_j}^{lo} \setminus I_{v_j}^{lomax}) \wedge v_k \in anc(v_p) \wedge v_k \in V_i\} \tag{50}$$

Sustainability: We claim that the above analysis of the WCRT bound for single-DAG systems is sustainable [19], i.e., when a node executes less than its WCET, the R_i will not exceed the computed worst-case bound. In the constructed analysis, the WCRT is obtained by finding the worst-case finish time of each node in a DAG. According to the Equations 41 and 42, the worst-case finish time of each node consists of three factors: 1) the finish time of the predecessor, 2) the associated intra-task delay, and 3) the WCET of the node.

Lemma 9. *The constructed analysis is sustainable if the intra-task delay is not increased when the predecessors of a node v_j execute less than their WCETs.*

Proof. We focus on one factor at a time to prove this lemma. First, if v_j itself executes less than its WCET, it can only result in a smaller Ft_{v_j}. Second, without considering the intra-task delay, given that the predecessors of v_j execute

less than their WCETs, then Ft_{v_j} can only demonstrate a monotonically non-increasing trend. Therefore, any sustainability issue in the constructed analysis could only occur when the predecessors of v_j execute less than their WCET but have caused an increased intra-task delay. □

Theorem 3. *The constructed WCRT bound is sustainable.*

Proof. Following Lemma 9, we focus on proving that for a node v_j, its intra-task delay will not increase when its predecessor nodes execute less than their WCETs. First, the set of concurrent nodes of v_j will remain the same regardless of the actual execution time v_j takes, which is derived purely based on execution dependency. Then, the list $I_{v_j}^{remove}$ cannot be affected as the nodes are accounted for based on the structural characteristics of the DAG instead of the execution time of the nodes (Theorem 2). This precludes $I_{v_j}^{potential}$ from being affected (Equation 45). To this end, we proved that the list $I_{v_j}^{potential}$ cannot be affected by a lower execution time of nodes in a DAG. Based on Equation 46, the intra-task delay is determined by $I_{v_j}^{hi}$, $I_{v_j}^{lomax}$, and $I_{v_j}^{lowpre}$. However, as shown in Equations 47, 49, and 50, these factors are computed based on the priorities and the WCETs of the nodes, which provide the maximum possible bound. Therefore, any node with a lower execution time will not increase the intra-task interference of v_j, and hence, the theorem follows. □

5.1.2 RTA for Multi-DAG Systems

In this section, we extend the analysis from a single DAG to a multi-DAG system, in which there exists a set of recurrent DAG tasks $\Gamma = \{\tau_1, \tau_2, ..., \tau_z\}$. The WCRT of τ_i is denoted as R_i. To calculate R_i, the goal is to determine the worst-case finish time of each node in τ_i (i.e. $Ft(v_{i,j}), \forall v_{i,j} \in V_i$) which follows the same principle as Section 5.1.1. The only difference is that each node will incur delay not only from nodes within the same DAG but also from other DAGs, i.e., inter-

task delay. To compute the delay, the identification of the concurrent nodes (see Equation 1) is updated to include nodes from other DAG tasks, denoted as $S^*(v_{i,j})$ for the multi-DAG case in Equation 51.

$$S^*(v_{i,j}) = S(v_{i,j}) \cup \bigcup_{\tau_k \in \Gamma} \left\{ \left\lceil \frac{R_i}{T_k} \right\rceil * V_k \middle| v_{i,j} \notin V_k \right\} \tag{51}$$

As shown in this equation, the set of concurrent nodes of $v_{i,j}$ in a multi-DAG system contains 1) the concurrent nodes from the same DAG (i.e. $S(v_{i,j})$ in Equation 23 and the nodes from other DAGs, which is denoted as $\bigcup_{\tau_k \in \Gamma} \{ \lceil \frac{R_i}{T_k} \rceil * V_k | v_{i,j} \notin V_k \}$. Notation $\lceil \frac{R_i}{T_k} \rceil$ provides the number of releases of τ_k during R_i and V_k represents all nodes in τ_k. The multi-DAG analysis follows the same process as the single-DAG analysis described in Section 5.1.1 but includes extra inter-task delay imposed by other DAG tasks. The key principle is still to work out the worst-case execution scenario of each node within a DAG.

Unlike the RTA for DAG tasks proposed in [86], as introduced in Section 2.5.3, which bounds the WCRT of DAG by calculating the intra-task and inter-task delays separately and incurs repetitive calculations of the delay leading to Limitation 8, the proposed RTA places all concurrent nodes in Equation (51). It analyzes intra-task and inter-task holistically, proceeding step by step (as shown in Section 5.1.1) to more precisely capture the actual intra-task and inter-task delays, effectively addressing the limitation.

5.2 Evaluation

In this section, we compare the tightness of the analytical bounds between the proposed and the state-of-the-art methods. The evaluation focuses on comparing the WCRT length (i.e. makespan) for single-DAG systems and the overall system schedulability of multi-DAG systems.

Experimental Setup.

We consider a homogeneous architecture with varied core number $m = [2, 32]$. Each DAG task in the evaluation is generated using two structural parameters: the *Length* indicating the number of node layers required during generation, and the *Parallelism* defining the number of nodes to be generated in a layer. Starting from a single source node, nodes in a DAG task are generated layer by layer given the *Length* and the *Parallelism* settings, where the node generation is finished by adding a single sink node. Each newly-generated node has a probability of 50% to be connected to existing ones. Finally, nodes without any predecessor (or successor) are directly connected to the source (or the sink) node.

In the single DAG evaluation, 1000 DAGs are generated under each system setting to compare the length of makespan. Each task is generated with $Parallelism \in [6, 16]$ and $Length \in [3, 13]$. The period of each DAG τ_i is randomly determined within the range of $T_i \in [1000ms, 2000ms]$, and its deadline is set equal to the period $D_i = T_i$. The utilization task is set to $U_{\tau_i} = 0.5$.

For the multi-DAG evaluation, 1000 systems are generated under each system setting to compare the percentage of schedulable systems. Each system contains $z = 8$ DAGs. We set $T_i \in [1000ms, 3000ms]$ and $D_i = T_i$. The total utilization of the system $\sum U$ is determined by multiplying a utilization factor in the range of [0.1, 0.35] by z, ensuring that the average utilization per core $\frac{\sum U}{m} < 1$. The individual utilization of each DAG τ_i is computed using the UUnifast Algorithm [14].

With utilization determined, the total WCET of τ_i is calculated by $C_{\tau_i} = U_{\tau_i} \times T_i$. The WCET of each node (e.g., C_{v_i} for a node v_i) is then randomly distributed by C_{τ_i}, where we enforce $C_{v_i} \geq 0$.

In addition, the intra-task priority assignment in [54] is applied for nodes in the DAG. The Rate Monotonic Priority Assignment is used to assign the DAG-level priorities.

The competing methods are illustrated below.

- The state-of-the-art generic bound for the limited preemption scheme in [86] (denoted as *Serrano2016*).

- The state-of-the-art priority-explicit bound proposed in [54]. In single-DAG, we include the intra-task blocking (i.e., $I_{v_j}^{lo}$ and $I_{v_j}^{lopre}$ in Equation 49 and 50). For multi-DAG, the inter-task blocking in [86] is applied [2]. (denoted as *He2021*).

- The RTA constructed in this chapter (denoted as *ours*).

5.2.1 Evaluation for Single-DAG Systems

Figures 27 to 29 present the makespan of a single DAG with varied m, degrees of parallelism, and lengths, respectively. Each makespan of a DAG is normalized against the maximum makespan observed in the experiment out of 1000 DAGs (where applied). A box-and-whisker plot is used to display the results, providing a standardized method for representing the distribution of a dataset. The central box spans the interquartile range, extending from the first quartile to the third quartile, capturing the middle 50% of the data. A horizontal line within this box indicates the median, signifying the data's central tendency. The top and bottom lines of the plot extend from the box to the dataset's minimum and maximum values, giving a comprehensive view of the data's range.

From Figure 27, we can observe that the proposed analysis demonstrates a constant lower bound on makespan compared to other methods. For instance, the proposed analysis outperforms He2021 and Serrano2016 by an average of 8.8% and 8.47% under $m = 6$, respectively. The reason for this observation is that the proposed analysis explicitly considers the parallel execution of nodes in

[2]We note that this modification is necessary for the application on a limited preemption scheduler and the modified analysis is still the state-of-the-art analysis with explicit priorities under the target scheduling scheme.

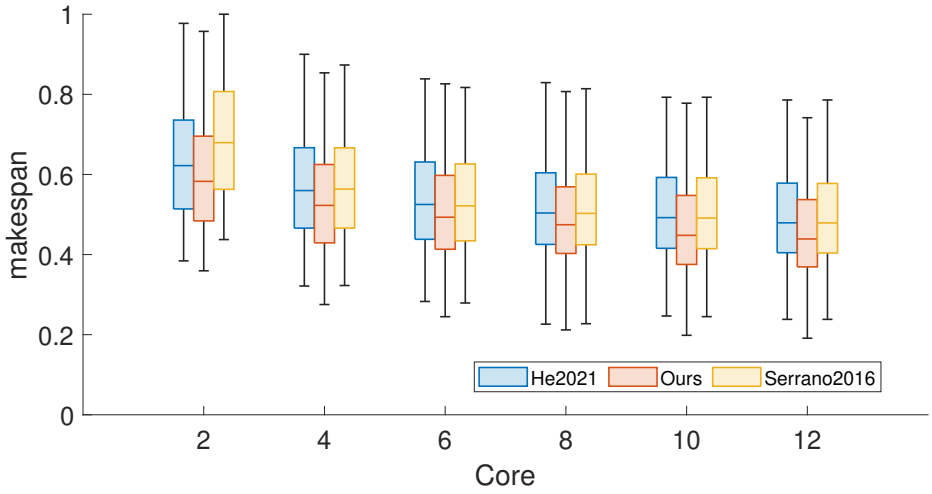


Figure 27: The makespan of a single DAG with varied m, Parallelism = 8 and Length = 7.

a DAG and only takes ones that can cause a delay, and hence, leads to lower intra-task delay.

Similar observations are also obtained in Figures 28 and 29, in which the proposed analysis is constantly better than the competing methods in terms of achieving tighter bounds on the makespan. From these figures, a notable observation is that the proposed analysis performs better when the length of the DAG is relatively low, e.g., with a length less than seven in Figure 29. This is because with a shorter length, fewer layers of nodes are generated so that less nodes will be included in $I_{v_j}^{potential}$. According to Equation 46, if nodes in $I_{v_j}^{potential}$ cannot take up all the cores, v_j can start execution without incurring any interference. However, in the state-of-the-art methods, such workload is still accounted for when bounding the intra-task delay, and hence, results in a longer makespan.

In addition, to offer a more detailed comparison of the evaluated methods, Tables 13 and 14 present a comprehensive dataset extracted from Figure 27

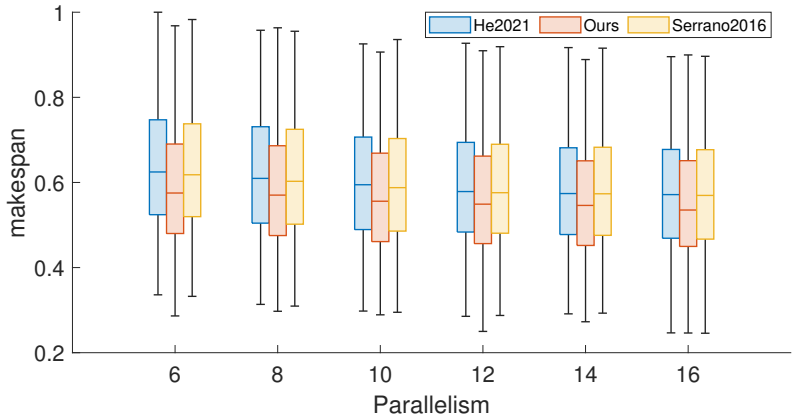


Figure 28: The makespan of a single DAG with varied degree of parallelism, $m = 6$ and Length $= 7$.

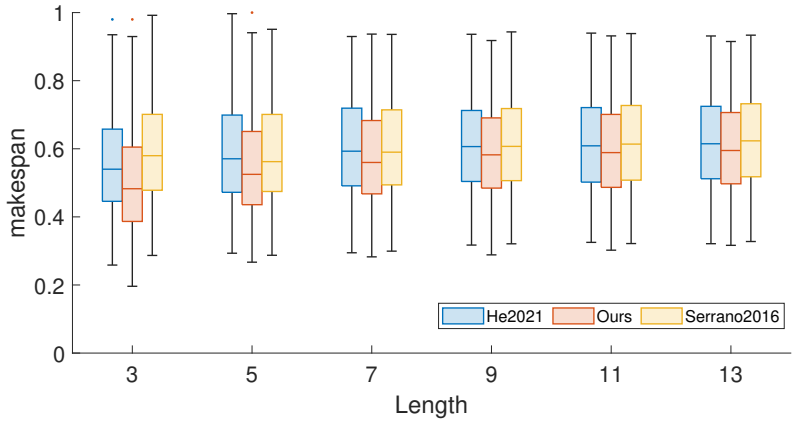


Figure 29: The makespan of a single DAG with varied length, $m = 6$, and Parallelism $= 12$.

Table 13: The Comparison of Makespan with He2021.

	Our $\succ$ He2021						Our $\prec$ He2021					
	m=2	m=4	m=6	m=8	m=10	m=12	m=2	m=4	m=6	m=8	m=10	m=12
number	981	995	991	992	1000	1000	5	5	9	8	0	0
avg.	7.21	9.17	8.88	9.24	8.04	6.75	0.06	1.65	1.52	1.86	0	0
max.	21.64	27.68	26.47	22.14	23.91	18.22	0.3	3.4	2.44	3.84	0	0

Table 14: The Comparison of Makespan with Serrano2016.

	Our $\succ$ Serrano2016						Our $\prec$ Serrano2016					
	m=2	m=4	m=6	m=8	m=10	m=12	m=2	m=4	m=6	m=8	m=10	m=12
number	1000	987	980	991	1000	1000	0	13	20	9	0	0
avg.	13.68	9.11	8.47	8.94	7.91	6.69	0	1.3	1.92	1.77	0	0
max.	28.29	26.75	22.01	20.56	19.28	16.87	0	3.06	5.21	3.84	0	0

with varied m. This dataset provides a comparison of the length of computed makespan between competing methods. For instance, $Our \succ He2021$ indicates cases where *Our* outperforms *He2021*, while $Our \prec He2021$ denotes the opposite. The notation **number** represents the number of advantageous or disadvantageous DAG tasks out of 1,000 generated tasks. **avg.** provides the average advantageous performance, and **max.** denotes the maximum advantageous performance. Taking the first column of Table 13 as an example, when $m = 2$, our bound computes 981 shorter makespan out of 1000 DAG tasks. In the 981 cases, we outperform *He2021* by 7.21% (i.e. shorter by 7.21%) in average. The maximum advantage in these cases is 21.64%.

From the tables, we can observe that the proposed analysis outperforms other schemes in most cases across all settings, and obtains an improvement of up to 9.24% and 13.68% on average, compared to He2021 and Serrano2016 respectively. However, we also observed that there exist cases where He2021 or Serrano2016 can produce a lower makespan bound. This is because in order to provide a safe bound, the $I_{v_j}^{lopre}$ in Equation 46 is introduced in our analysis to avoid over-optimistic computations when bounding the blocking delay from low-priority nodes. However, such situations rarely occur and the proposed analysis demonstrates a significant advantage over the competing analysis in almost all cases.

5.2.2 Evaluation for Multi-DAG Systems

Figures 30 and 31 present the schedulability of the evaluated methods in multi-DAG systems, with a varied number of cores m and utilization per core $\frac{\sum U}{m}$.

In the multi-DAG case, the proposed analysis demonstrates the most pronounced advantages, in which the schedulability of other methods drops quickly and schedules few systems when either the $m < 14$ or the $\frac{\sum U}{m} > 0.2$. In contrast, the proposed analysis can still schedule up to 100% of the systems in majority cases (e.g., $m = 12$ in Figure 30 and with $\frac{\sum U}{m} = 0.5$ in Figure 31). This observation is obtained due to the combined efforts of the proposed RTA in reducing the bound of both the intra-task and inter-task delays.

More specifically, in the single-DAG analysis, we focus on calculating the worst-case scenario of nodes instead of paths which allow a finer-grained delay analysis. The key factor for better schedulability is the removal of unnecessary interference and blocking delay that we summarized in Equation 44 and the condition that judges whether delay will occur in Equation 46. Moreover, in the multi-DAG analysis, we continue to analyze the worst-case execution scenario of each node and treat other DAGs as concurrent nodes as shown in Equation 51 which allows the inter-task delay to be capured precisely without much redundancy. However, He2021 and Serreno2016 always compute the interference delay by averaging the total workload of the interfering DAGs or nodes, and include repetitive calculations when bounding the blocking delay (See Limitation 8 discussed in Section 2.5.3).

We also observed that *He2021* and *Serreno2016* have almost the same schedulibility in Figures 30 and 31. The reason is that the work in [54] provides a bound for single-DAG preemptive scheduling which can have an obvious advantage over generic bound because only high-priority nodes are accounted as interference. However, we introduce non-preemptive (our focus) features to *He2021* which in-

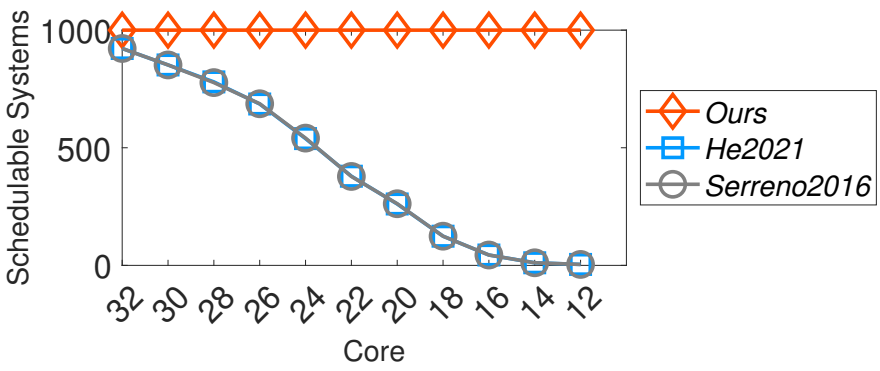


Figure 30: The system schedulability for multi-DAGs under a varied m, Parallelism = 6, and Length = 6.

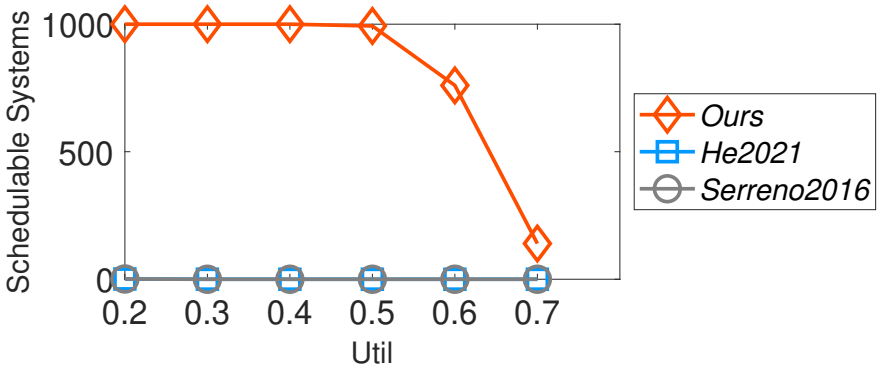


Figure 31: The system schedulability for multi-DAGs under varied $\frac{\sum U}{m}$, $m = 4$, Parallelism = 6, and Length = 6.

cludes low-priority blocking delay, and the advantage becomes trivial. Moreover, *He2021* inherits the analysis of Serreno2016 in multi-DAG systems, hence their overall difference is negligible with a limited-preemption scheduling scheme.

5.3 Summary

This chapter focuses on the RTA of DAG tasks operating under a global limited preemption scheme. We introduce novel RTA analysis techniques tailored for both single-DAG and multi-DAG systems. For the single-DAG analysis, we meticulously address intra-task delays. This is achieved by identifying and eliminating infeasible delays, guided by an in-depth exploration of a DAG's dependency structure. Furthermore, we capitalize on the inherent parallelism among nodes, examining conditions where nodes can concurrently execute without inducing any delay, thereby addressing Limitation 7 as stated in Section 2.5.3.

Diverging from conventional methods that formulate the multi-DAG RTA by simply aggregating the inter-task delay to the single-DAG RTA. Our approach centers on constructing the WCRT by determining the worst-case finish time for each node within a DAG. Such a methodology grants enhanced flexibility in bounding the WCRT for multi-DAG scenarios. To derive the WCRT for multi-DAG systems, we also compute the worst-case finish time for each node, treating other DAG tasks as parallel entities. Subsequently, we revisit the analysis process to holistically ascertain a more accurate interference and blocking delay, eliminating the redundancy in intra-task and inter-task delay calculations which addresses the Limitation 8 as summarized in Chapter 2.5.3.

Our experimental results distinctly highlight the advanced efficacy of our analysis in comparison to prevailing methods. Comprehensive experiments reveal that, when juxtaposed with the techniques in [86] and [54], our analysis yields considerably tighter bounds, amplifying system schedulability by over

300%. Notably, our approach demonstrates pronounced superiority in systems facing intense schedulability constraints, achieving scheduling for up to 100% of such systems, a feat unattained by competing methodologies. The contributions of this chapter established the success criteria SC-5 of the thesis.

Chapter 6

Conclusion

In this thesis, we undertake a comprehensive investigation into three pivotal research areas in real-time systems: reliable resource sharing in Mixed-Criticality Systems (MCS), resource-aware task allocation methods, and precise Response Time Analysis (RTA) for tasks modeled as Directed Acyclic Graphs (DAG). These areas are fundamental to enhancing the performance of modern real-time embedded systems.

Chapter 2 serves as the foundation of this thesis, providing essential background knowledge on real-time task models, scheduling schemes, and schedulability analysis—all crucial for ensuring timing predictability. This chapter delves into the intricacies of shared resources and MCS models and introduces advanced RTA methods that incorporate both shared resources and MCS. It addresses limitations 1-8 of current research bottlenecks, setting the stage for the solutions proposed in the subsequent chapters.

In Chapter 3, we introduce a novel fault-tolerance solution to ensure reliable resource sharing in MCS. This solution encompasses a system execution model and the Multiprocessor Stack Resource Sharing Protocol Fault Tolerance (MSRP-FT) protocol—a novel approach designed to minimize blocking time during critical sections. By leveraging remote processors to support the resource-holding task, this solution enhances fault tolerance and provides a timing guarantee through a comprehensive RTA.

Chapter 4 presents a resource-aware task allocation method for preemptive, fully-partitioned systems scheduled by Fixed-Priority Scheduling (FPS) with First-In-First-Out (FIFO) spin locks applied. This method transcends the limi-

tations of traditional approaches by using a Resource Contention Model (RCM) to approximate resource contention between tasks. This guides the allocation process to diminish resource competition and task blocking, ultimately improving system performance.

Chapter 5 zeroes in on periodic DAG tasks in homogeneous multiprocessor systems, introducing a novel priority-explicit RTA. This chapter unveils new analysis techniques that sidestep complex computations, offering tighter upper bounds for DAG tasks. It achieves this by accurately bounding both intra-task and inter-task delays, laying the groundwork for the application of RTA in the modern embedded systems with increasing complexities and stringent requirements.

6.1 Contributions

In this thesis, we introduce a High-Performance Real-time Scheduling (HPRTS) framework designed to tackle the existing limitations in crucial areas of real-time systems. These areas encompass reliable resource sharing, contention-aware allocation, and RTA for multi-DAGs in multiprocessor architectures, all of which have the potential to impede the advancement of current real-time systems toward higher performance.

In Chapter 2, we provide a literature review to contextualize the research of this thesis.

- We demonstrate a deep understanding of the field by critically analyzing and synthesizing a wide range of relevant literature, theories, and methodologies.
- We identify gaps in existing studies, summarizing Limitations 1 to 8.

- We establish through Lemma 3 that the existing RTA bound for DAG tasks as presented in [108] is not secure.

In chapter 3, to guarantee reliable resource sharing in multiprocessor MCS, we propose a fault-tolerance solution:

- We construct a system execution model compatible with an arbitrary number of criticality levels. In our model, faults occurring in normal sections and critical sections are treated separately.

- A novel protocol, MSRP-FT, is proposed to address faults during critical sections, aiming to minimize blocking time.

- An holistic RTA, building on the state-of-the-art analysis is developed to provide timing guarantees for the proposed solution.

Compared to the existing methods in the same system context, our approach improves system schedulability by up to 151.5% in average. These contributions address Limitations 1, 2 and 3 as illustrated in Chapter 2 and fulfill the success criteria SC-1 and SC-2 of this thesis.

In chapter 4, to provide effective task allocation methods with the presence of shared resources, we present a resource-aware task allocation method for fully-partitioned systems scheduled by FPS with FIFO-spin locks applied:

- We develop a RCM to approximate resource contention between tasks, by anlayzing resource usage and potential blocking.

- A resource-aware task allocation algorithm is developed to allocate tasks with the highest resource contention to the same processor, which reduces global resource competition and task blocking.

We initiated evaluations on homogeneous architectures. In most test cases, our method outperformed other prevailing techniques. Moreover, in scenarios

particularly favorable to our approach, we surpassed the search-based method (Resource-Oriented Partitioning (*ROP*)) by an average of 74% in schedulability. Additionally, the computational overhead of *ROP* was, on average, 5.9 times that of our method. Upon transitioning to the more intricate heterogeneous architectures with diverse core frequencies, the superiority of our method was further underscored, especially in comparison to numerous other techniques. Our contributions in Chapter 4 address Limitations 4, 5 and 6 as stated in Chapter 2 and fulfill the success criteria SC-3 and SC-4 of this thesis.

In chapter 5, for multi-core systems with multi-DAG tasks and a global limited preemption scheme, we propose a novel priority-explicit RTA:

- We provide a new analysis technique for a single DAG that explores node-level parallelism and safely removes unnecessary workload when bounding intra-task delays.
- We also provide a new analysis method for multi-DAGs which analyze the intra- and inter-task delays holistically to avoid redundant calculations.

Extensive experiments demonstrate that the constructed analysis achieves tighter bounds and can improve system schedulability by over 300%. It has an even larger advantage for systems with high schedulability pressure, scheduling up to 100% of such systems compared to 0% for competing methods. This developed RTA for single- and multi-DAG tasks addresses Limitations 7 and 8 as stated in Chapter 2 and fulfills the success criteria SC-5 of the thesis.

Revisiting the hypothesis of this thesis, we posited that the proposed HPRTS framework could achieve promising performance improvements in reliable resource-sharing, resource-aware task allocation, and RTA for multi-DAG systems. Reflecting on the contributions and experimental results presented in this thesis: Our fault-tolerant solution effectively addresses the challenges of resolving faults in MCS with shared resources, thereby enhancing system schedulability. The

resource-aware task allocation algorithm we introduced reduces resource contention, outperforming existing methods. Furthermore, our RTA for multi-DAG systems provides a tighter WCRT bound, adeptly addressing the challenges of hardware resource over-provisioning. Consequently, the initial hypothesis has been validated.

6.2 Future Research

In this subsection, we will outline potential future research directions building upon the current strategies that ensure reliable resource-sharing, resource-aware task allocation methods, and timing predictability for the scheduling of multi-DAG systems. Building on the existing research, we aim to extend to a more complex system model or to develop advanced methods that are more aligned with the applications of real-world systems. This approach will further facilitate the optimization of real-time systems, steering them towards higher performance.

Fault-Tolerance Approaches with Different Protocols

In Chapter 3, we introduce the novel fault-tolerance protocol MSRP-FT, which operates based on the non-preemptive FIFO-spin characteristics of MSRP. In this setup, tasks are placed in a FIFO queue and start to spin non-preemptively, engaging in busy-wait states for the locks on their host core. These tasks are utilized to assist the resource holder in executing the associated critical section in parallel, thereby addressing potential faults.

The core principle of this fault-tolerant approach for critical sections is to repurpose the wasted cycles of the spinning tasks in the FIFO queue. This strategy ensures reliability for each critical section during a single access, aiming to reduce the time spent on fault tolerance and resource contention. Meanwhile, the simple non-preemptive FIFO-spin characteristics offer a naturally suitable plat-

form for the application of MSRP-FT. This allows for the utilization of wasted cycles and prevents helping tasks from being interrupted midway while assisting the resource holder. However, the non-preemptive characteristics are not as accommodating to local high-priority tasks, which often face more urgent deadlines but can experience intensive blocking from low-priority tasks whenever a resource request is made.

Looking ahead, our future research will apply the proposed fault-tolerance approaches to other preemptive FIFO-spin-based multiprocessor resource sharing protocols, such as Multiprocesser resource sharing Protocol (MrsP), cited in [22]. Under this approach, tasks inherit a ceiling when requesting shared resources, meaning they can be preemptive while spinning for shared resources. Implementing MrsP necessitates the development of more detailed execution strategies to respond to faults when they occur, especially when the resource holder and helping tasks are preempted by a local task. In line with the developed approaches, new RTA will be introduced to enhance the system's timing predictability.

Include Overheads in Fault-Tolerance Approach

In Chapter 3, the proposed fault-tolerance approaches are undertaken with the assumption that no overheads are explicitly considered. In revising the MSRP-FT, the task at the head of the FIFO queue will access the shared resource. The code segment to be executed by the head task and the internal states (e.g., variables) of the resource are replicated according to the number of tasks in the FIFO queue. Subsequently, replicas are stored in the local memory of each core. Each task in the FIFO queue, including the head task, executes a replica on their host cores in parallel and updates the results on the local replica independently. Once a correct output is produced, it will update the global resource, and the execution of other replicas will be signaled to stop and abandon the replicas.

The implementation of MSRP-FT involves many steps, including fetching the shared resources, replicating the shared resources, and facilitating communication between tasks when updating the shared resources. It also encompasses the context switch of helping tasks when assisting different resource holders to execute the critical sections. While some of these overheads may be negligible, they are still worth exploring. In future research, we will analyze the potential overheads at each step of MSRP-FT, incorporating them into the RTA of MSRP-FT to more precisely evaluate its performance. This will help determine the scenarios and extent to which the overheads can impact the effectiveness of MSRP-FT.

Develop a RCM Model based on other Lock Mechanisms

In Chapter 4, the RCM delineates the level of resource contention occurring between tasks operating under FIFO-Spin-based resource sharing protocols. This contention level is ascertained approximately through a meticulous analysis of the aggregate demand of tasks for resources, coupled with the regulatory stipulations inherent in FIFO-Spin-based resource sharing protocols, thereby facilitating a more precise computation of potential task blockages.

Nevertheless, as elucidated in [104], resource sharing protocols in multiprocessor environments inherently possess both advantages and disadvantages, with no single protocol unequivocally surpassing others. Looking forward, our forthcoming research endeavors will encompass the expansion of the prevailing resource contention model to accommodate a diverse array of resource sharing protocols. This initiative aims to accurately depict task resource-access behaviors governed by the distinct regulations of various resource sharing protocols. Consequently, this will empower system architects in the realm of modern embedded systems with a broader spectrum of design alternatives and enhanced adaptability, given

that different protocols are optimally suited for disparate systems

Develop Resource-Aware Task Allocation on Heterogeneous Architecture

In Chapter 4, we evaluate the RCM and resource-aware task allocation algorithms in both homogeneous and heterogeneous architectures. However, the current assumed heterogeneous model represents an initial or simple heterogeneous architecture. Cores are presumed to have Dynamic Voltage and Frequency Scaling (DVFS)-enabled and to operate at different frequencies during runtime. Moreover, they do not exhibit markedly different execution abilities, meaning that tasks can execute on any core in the system.

In the existing heterogeneous setup, the proposed allocation algorithms based on the RCM can still yield outstanding results with the current design. Looking forward to future research, we plan to explore more complex heterogeneous architectures, wherein different types of tasks can only be executed on specified processors. In such scenarios, the contention model and allocation algorithms must be redeveloped to consider not only task contention and resource frequencies but also to accommodate different task types. Balancing these elements will introduce more complex research problems, yet it is essential for enhancing the performance of high-end real-time systems.

Developing RTA for Typed-DAG Tasks

In Chapter 5, we introduce a new analysis technique for single and multi-DAGs that leverages node-level parallelism and safely eliminates unnecessary blocking workload. The proposed RTA is grounded in simple task models within a homogeneous multiprocessor architecture, where nodes within DAG tasks can execute on different cores.

However, emerging research has begun to focus on typed-DAG tasks, wherein nodes within a DAG are required to execute on different cores. These typed-DAG tasks can represent a wider range of systems, as applications stemming from different components of the platform may have vastly different execution requirements, yet still maintain execution dependencies. Consequently, future research will analyze typed-DAG tasks on heterogeneous architectures under a steadfast scheduling paradigm, continuing to consider the parallelism between nodes while developing an effective RTA that ensures safety and reduces pessimism.

Discover Resource-Sharing of DAG Tasks

The current focus of this thesis in Chapter 5 is on simple DAG task models without the consideration of shared resources. However, this scenario will eventually need to be addressed, as shared resources are prevalent in modern real-time embedded systems.

In future research on the topic of DAG task models, we will take into account the presence of resource sharing, not only between DAG tasks but also between nodes within a DAG. In this scenario, it will be essential to explore the effectiveness of different resource sharing protocols, ensuring that node scheduling satisfies dependency constraints while also maintaining the data integrity of shared resources. Moreover, the RTA of such systems needs to be developed to guarantee both timing constraints and dependency constraints of tasks. New research challenges will arise under the complicated research background, such as determining the blocking delay, which will depend not only on the execution time of nodes but also on the amount of time each node spends acquiring shared resources.

6.3 Concluding Remarks

In the realm of academic research, the domain of real-time systems consistently presents a myriad of intricate avenues for exploration and innovation. The comprehensive findings and methodological contributions presented in this thesis not only elucidate the current state of the art but also establish a rigorous foundation for subsequent scholarly endeavors. As we look forward, it becomes imperative to build upon this foundation with the overarching objective of further enhancing the efficacy, robustness, and dependability of real-time embedded systems, ensuring they meet the evolving demands of contemporary applications.

List of Abbreviations

HPRTS: High Performance Real-time Scheduling

MCS: Mixed-Criticality Systems

RCM: Resource Contention Model

I/O: Input/Output

DAG: Directed Acyclic Graph

RTA: Response Time Analysis

QoS: Quality of Service

ABS: Anti-lock Braking System

WF: Worst-Fit

FF: First-Fit

BF: Best-Fit

NF: Next-Fit

FPS: Fixed-Priority Scheduling

EDF: Earliest Deadline First

LLF: Least Laxity First

NP: Non-Preemptive

WCET: Worst-Case Execution Time

WCRT: Worst-Case Response Time

FIFO: First-in-First-out

OPCP: Original Priority Ceiling Protocol

IPCP: Immediate Priority Ceiling Protocol

PCP: Priority Ceiling Protocol

MSRP: Multiprocessor Stack Resource Protocol

AMC: Adaptive Mixed Criticality

HLC-PCP: Highest-Locker Criticality, Priority-Ceiling Protocol

ILP: Integer Linear Programming

SPA: Synchronization-Aware Partitioning Algorithm

ROP: Resource-Oriented Partitioning

CF: Contention Factor

URLLC: Ultra-Reliable Low Latency Communications

MrsP: Multiprocessor Resource Sharing Protocol

MSRP-FT: Multiprocessor Stack Resource Protocol Fault Tolerance

DVFS: Dynamic Voltage and Frequency Scaling

References

[1] Z. Al-bayati, J. Caplan, B. H. Meyer, and H. Zeng. A four-mode model for efficient fault-tolerant mixed-criticality systems. In *IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016.

[2] M. Alfranseder, M. Deubzer, B. Justus, J. Mottok, and C. Siemers. An efficient spin-lock based multi-core resource sharing protocol. In *IEEE International Performance Computing and Communications Conference (IPCCC)*, 2014.

[3] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[4] N. C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. In *Technical Report YCS 164*. University of York, Department of Computer Science, 1991.

[5] AUTOSAR. AUTOSAR Adaptive Platform, November 2019.

[6] T. P. Baker. An analysis of edf schedulability on a multiprocessor. *IEEE transactions on parallel and distributed systems*, 16(8):760–768, 2005.

[7] T. P. Baker. A comparison of global and partitioned edf schedulability tests for multiprocessors. *22nd IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 10 pp.–, 2005.

[8] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, M. Peri, and S. Pezzini. Fault-tolerant platforms for automotive safety-critical ap-

plications. In *ACM Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2003.

[9] S. Baruah. The federated scheduling of systems of conditional sporadic DAG tasks. In *International Conference on Embedded Software*, pages 1–10, 2015.

[10] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie, and A. Wiese. A generalized parallel task model for recurrent real-time processes. In *Real-Time Systems Symposium*, pages 63–72, 2012.

[11] S. K. Baruah, A. Burns, and R. I. Davis. Response-time analysis for mixed criticality systems. In *IEEE Real-Time Systems Symposium (RTSS)*, 2011.

[12] M. Bertogna, G. Buttazzo, and G. Yao. Improving feasibility of fixed priority tasks using non-preemptive regions. In *2011 IEEE 32nd Real-Time Systems Symposium*, pages 251–260. IEEE, 2011.

[13] E. Bini and G. Buttazzo. Schedulability analysis of real-time systems with stochastic task execution times. *IEEE Transactions on Computers*, 53(11):1462–1473, 2005.

[14] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Springer Real-Time Systems (RTS)*, 2005.

[15] B. B. Brandenburg. Multiprocessor real-time locking protocols: A systematic review. *arXiv preprint arXiv:1909.09600*, 2019.

[16] I. Broster and A. Burns. A practical solution to the problem of schedulability analysis with overrun without abort. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS)*, pages 209–218. IEEE, 2003.

[17] A. Burns. The application of the original priority ceiling protocol to mixed criticality systems. *Proc. ReTiMiCS, RTCSA*, 2013.

[18] A. Burns and S. Baruah. Towards a more practical model for mixed criticality systems. In *Workshop on Mixed-Criticality Systems (colocated with RTSS)*, 2013.

[19] A. Burns and S. K. Baruah. Sustainability in real-time scheduling. *Journal of Computing Science and Engineering*, 2(1):74–97, 2008.

[20] A. Burns and R. I. Davis. Mixed criticality systems—a review. *Department of Computer Science, University of York, Tech. Rep*, 2019.

[21] A. Burns and A. J. Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001.

[22] A. Burns and A. J. Wellings. A schedulability compatible multiprocessor resource sharing protocol–mrsp. In *IEEE Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2013.

[23] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer Science & Business Media, 2011.

[24] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo. Partitioned fixed-priority scheduling of parallel tasks without preemptions. In *IEEE Real-Time Systems Symposium*, pages 421–433. IEEE, 2018.

[25] D. Cederman, B. Chatterjee, N. Nguyen, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas. A study of the behavior of synchronization methods in commonly used languages and systems. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 1309–1320. IEEE, 2013.

[26] G. Chen, N. Guan, K. Huang, and W. Yi. Fault-tolerant real-time tasks scheduling with dynamic fault handling. *Elsevier Journal of Systems Architecture (SA)*, 2020.

[27] N. Chen, S. Zhao, I. Gray, A. Burns, S. Ji, and W. Chang. MSRP-FT: Reliable resource sharing on multiprocessor mixed-criticality systems. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 201–213. IEEE, 2022.

[28] P. Chen, W. Liu, X. Jiang, Q. He, and N. Guan. Timing-anomaly free dynamic scheduling of conditional DAG tasks on multi-core systems. *ACM Transactions on Embedded Computing Systems*, 18(5):1–19, 2019.

[29] 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.

[30] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM computing surveys (CSUR)*, 43(4):1–44, 2011.

[31] A. C. De Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, volume 18, pages 1–42, 2010.

[32] S. Di Cairano and I. V. Kolmanovsky. Real-time optimization and model predictive control for aerospace and automotive applications. In *2018 annual American control conference (ACC)*, pages 2392–2409. IEEE, 2018.

[33] R. Dobrin and G. Fohler. Reducing the number of preemptions in fixed priority scheduling. In *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.*, pages 144–152. IEEE, 2004.

[34] P. Ekberg and M. Törngren. Reliability in mixed-criticality systems. *Proceedings of the 2nd Workshop on Analytic Virtual Integration of Cyber-Physical Systems*, 2012.

[35] P. Emberson, R. Stafford, and R. I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, pages 6–11, 2010.

[36] J. Fonseca, G. Nelissen, and V. Nélis. Improved response time analysis of sporadic DAG tasks for global FP scheduling. In *International Conference on Real-Time Networks and Systems*, pages 28–37, 2017.

[37] J. Fonseca, G. Nelissen, V. Nelis, and L. M. Pinho. Response time analysis of sporadic DAG tasks under partitioned scheduling. In *Symposium on Industrial Embedded Systems*, pages 1–10, 2016.

[38] J. C. Fonseca, V. Nélis, G. Raravi, and L. M. Pinho. A multi-DAG model for real-time parallel applications with conditional execution. In *Annual ACM Symposium on Applied Computing*, pages 1925–1932, 2015.

[39] K. Funaoka, S. Kato, and N. Yamasaki. Work-conserving optimal real-time scheduling on multiprocessors. In *2008 Euromicro Conference on Real-Time Systems*, pages 13–22. IEEE, 2008.

[40] S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa, and K. Lange. AUTOSAR–a worldwide standard is on the road. In *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, volume 62, 2009.

[41] P. Gai, G. Lipari, and M. Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *IEEE Real-Time Systems Symposium (RTSS)*, 2001.

[42] P. Gai, G. Lipari, and M. Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Real-Time Systems Symposium (RTSS)*, pages 73–83. IEEE, 2001.

[43] Y. Gao and P. Zhang. A survey of homogeneous and heterogeneous system architectures in high performance computing. In *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, pages 170–175. IEEE, 2016.

[44] H. Gomaa. *Real-Time Systems Design and Analysis: An Engineer's Handbook*. John Wiley & Sons, 2004.

[45] R. L. Graham. Bounds on multiprocessing timing anomalies. *Journal on Applied Mathematics*, 17(2):416–429, 1969.

[46] F. E. Gruber. Industry 4.0: A best practice project of the automotive industry. In *Digital Product and Process Development Systems: IFIP TC 5 International Conference, NEW PROLAMAT 2013, Dresden, Germany, October 10-11, 2013. Proceedings*, pages 36–40. Springer, 2013.

[47] J.-J. Han, X. Tao, D. Zhu, and L. T. Yang. Resource sharing in multicore mixed-criticality systems: Utilization bound and blocking overhead. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2017.

[48] J.-J. Han, X. Wu, D. Zhu, H. Jin, L. T. Yang, and J.-L. Gaudiot. Synchronization-aware energy management for VFI-based multicore real-time systems. *IEEE Transactions on Computers*, 61(12):1682–1696, 2012.

[49] J.-J. Han, D. Zhu, X. Wu, L. T. Yang, and H. Jin. Multiprocessor real-time systems with shared resources: Utilization bound and mapping. *IEEE Transactions on Parallel and Distributed Systems*, 25(11):2981–2991, 2013.

[50] P. B. Hansen. *The origin of concurrent programming: from semaphores to remote procedure calls*. Springer Science & Business Media, 2013.

[51] M. A. Haque, H. Aydin, and D. Zhu. Real-time scheduling under fault bursts with multiple recovery strategy. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[52] M. Hatami. Semi-partitioned scheduling hard real-time periodic dags in multicores. In *The Proceeding of First Work-in-Progress Session of 2018 CSI International Symposium on Real-Time and Embedded Systems and Technologies*, page 9, 2018.

[53] Q. He, N. Guan, Z. Guo, et al. Intra-task priority assignment in real-time scheduling of dag tasks on multi-cores. *IEEE Transactions on Parallel and Distributed Systems*, 30(10):2283–2295, 2019.

[54] Q. He, M. Lv, and N. Guan. Response time bounds for dag tasks with arbitrary intra-task priority assignment. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[55] P.-C. Hsiu, D.-N. Lee, and T.-W. Kuo. Task synchronization and allocation for many-core real-time systems. In *Proceedings of the ninth ACM international conference on Embedded software*, pages 79–88, 2011.

[56] H.-M. Huang, C. Gill, and C. Lu. Implementation and evaluation of mixed-criticality scheduling approaches for sporadic tasks. *ACM Transactions on Embedded Computing Systems (TECS)*, 2014.

[57] W.-H. Huang, M. Yang, and J.-J. Chen. Resource-oriented partitioned scheduling in multiprocessor systems: How to partition and how to share? In *Real-Time Systems Symposium (RTSS)*. IEEE, 2016.

[58] S. Ijaz and E. U. Munir. Mopt: list-based heuristic for scheduling workflows in cloud environment. *The Journal of Supercomputing*, 75:3740–3768, 2019.

[59] I. ISO. 26262: Road vehicles-functional safety. *International Standard ISO/FDIS*, 2018.

[60] X. Jiang, N. Guan, X. Long, and W. Yi. Semi-federated scheduling of parallel real-time tasks on multiprocessors. In *Real-Time Systems Symposium*, pages 80–91, 2017.

[61] M. Klein, T. Ralya, B. Pollak, R. Obenza, and M. González Harbour. Static scheduling of hard real-time tasks. *Software Engineering Journal*, 8(3):116–128, 1993.

[62] K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *Real-Time Systems Symposium (RTSS)*, pages 469–478. IEEE, 2009.

[63] P. A. Laplante. *Real-time systems design and analysis*. John Wiley & Sons, 2004.

[64] H. Leinster. Deadline monotonic scheduling. *Real-Time Systems*, 20(3):255–274, 2001.

[65] J. Li, J. J. Chen, K. Agrawal, C. Lu, C. Gill, and A. Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *Euromicro Conference on Real-Time Systems*, pages 85–96, 2014.

[66] C.-C. Lin, M. Günzel, J. Shi, T. T. Seidl, K.-H. Chen, and J.-J. Chen. Scheduling periodic segmented self-suspending tasks without timing anomalies. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 161–173. IEEE, 2023.

[67] 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.

[68] J. W. S. Liu. *Real-time systems*. Prentice Hall, 2000.

[69] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[70] J. M. Lopez, J. Diaz, and D. Garcia. Utilization bounds for edf scheduling on real-time multiprocessor systems. *Real-Time Systems*, 28(1):39–68, 2004.

[71] J. M. Lopez, J. L. Diaz, and D. F. Garcia. A comprehensive comparison of partitioning schedules for real-time systems. In *20th IEEE Real-Time Systems Symposium (RTSS '00)*, pages 15–24. IEEE, 2000.

[72] R. Madhura, B. L. Elizabeth, and V. R. Uthariaraj. An improved list-based task scheduling algorithm for fog computing environment. *Computing*, 103:1353–1389, 2021.

[73] A. Meixner, M. E. Bauer, and D. Sorin. Argus: Low-cost, comprehensive error detection in simple cores. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007.

[74] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. C. Buttazzo. Response-time analysis of conditional dag tasks in multiprocessor systems. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 211–221. IEEE, 2015.

[75] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

[76] N. Muthuvelu, J. Liu, N. L. Soe, S. Venugopal, A. Sulistio, and R. Buyya. A dynamic job grouping-based scheduling for deploying applications with fine-grained tasks on global grids. In *Proceedings of the 2005 Australasian*

workshop on Grid computing and e-research-Volume 44, pages 41–48. Citeseer, 2005.

[77] J. Park. A deadlock and livelock free protocol for decentralized internet resource coallocation. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 34(1):123–131, 2004.

[78] R. M. Pathan. Fault-tolerant and real-time scheduling for mixed-criticality systems. *Springer Real-Time Systems (RTS)*, 2014.

[79] F. Pop, C. Dobre, and V. Cristea. Genetic algorithm for dag scheduling in grid environments. In *2009 IEEE 5th International Conference on Intelligent Computer Communication and Processing*, pages 299–305. IEEE, 2009.

[80] S. Punnekkat, A. Burns, and R. Davis. Analysis of checkpointing for real-time systems. *Springer Real-Time Systems (RTS)*, 2001.

[81] R. Rajkumar. *Synchronization in real-time systems: a priority inheritance approach*, volume 151. Springer Science & Business Media, 2012.

[82] E. Rosti, E. Smirni, G. Serazzi, and L. W. Dowdy. Analysis of non-work-conserving processor partitioning policies. In *Job Scheduling Strategies for Parallel Processing: IPPS'95 Workshop Santa Barbara, CA, USA, April 25, 1995 Proceedings 1*, pages 165–181. Springer, 1995.

[83] S. Safari, M. Ansari, G. Ershadi, and S. Hessabi. On the scheduling of energy-aware fault-tolerant mixed-criticality multicore systems with service guarantee exploration. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2019.

[84] M. Salehi, A. Ejlali, and B. M. Al-Hashimi. Two-phase low-energy n-modular redundancy for hard real-time multi-core systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2016.

[85] F. Samie, L. Bauer, and J. Henkel. Iot technologies for embedded computing: A survey. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 1–10, 2016.

[86] M. A. Serrano, A. Melani, M. Bertogna, and E. Quiñones. Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions. In *Design, Automation & Test in Europe Conference & Exhibition*, pages 1066–1071, 2016.

[87] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on computers (TC)*, 1990.

[88] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley, 2018.

[89] L. Spainhower and T. A. Gregg. Ibm s/390 parallel enterprise server g5 fault tolerance: A historical perspective. *IBM Journal of Research and Development*, 1999.

[90] J. A. Stankovic. Real-time and embedded systems. *ACM Computing Surveys (CSUR)*, 28(1):205–208, 1996.

[91] H. Takada and K. Sakamura. A novel approach to multiprogrammed multiprocessor synchronization for real-time kernels. In *IEEE Proceedings Real-Time Systems Symposium (RTSS)*, 1997.

[92] A. Thekkilakattil, R. I. Davis, R. Dobrin, S. Punnekkat, and M. Bertogna. Multiprocessor fixed priority scheduling with limited preemptions. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, pages 13–22, 2015.

[93] G. Upasani, X. Vera, and A. González. Setting an error detection infrastructure with low cost acoustic wave detectors. In *IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[94] G. Upasani, X. Vera, and A. González. Avoiding core's due sdc via acoustic wave detectors and tailored error containment and recovery. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2014.

[95] K. Vaidyanathan and K. S. Trivedi. Extended classification of software faults based on aging. In *Fast Abstract, Int. Symp. Software Reliability Eng., Hong Kong*. Citeseer, 2001.

[96] M. Verucchi, M. Theile, M. Caccamo, and M. Bertogna. Latency-aware generation of single-rate DAGs from multi-rate task sets. In *Real-Time and Embedded Technology and Applications Symposium*, pages 226–238, 2020.

[97] A. Wieder and B. B. Brandenburg. Efficient partitioning of sporadic real-time tasks with shared resources and spin locks. In *Symposium on Industrial Embedded Systems (SIES)*, pages 49–58. IEEE, 2013.

[98] A. Wieder and B. B. Brandenburg. On spin locks in AUTOSAR: Blocking analysis of FIFO, unordered, and priority-ordered spin locks. In *Real-Time Systems Symposium (RTSS)*, pages 45–56. IEEE, 2013.

[99] M. Yang, Z. Chen, X. Jiang, N. Guan, and H. Lei. DPCP-p: A distributed locking protocol for parallel real-time tasks. In *Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[100] M. Yang, W.-H. Huang, and J.-J. Chen. Resource-oriented partitioning for multiprocessor systems with shared resources. *IEEE Transactions on Computers*, 68(6):882–898, 2018.

[101] W. K. Youn, S. B. Hong, K. R. Oh, and O. S. Ahn. Software certification of safety-critical avionic systems: DO-178C and its impacts. *Aerospace and Electronic Systems Magazine*, 2015.

[102] W. K. Youn, S. B. Hong, K. R. Oh, and O. S. Ahn. Software certification of safety-critical avionic systems: Do-178c and its impacts. *IEEE Aerospace and Electronic Systems Magazine*, 30(4):4–13, 2015.

[103] Q. Zhao, Z. Gu, and H. Zeng. Hlc-pcp: A resource synchronization protocol for certifiable mixed criticality scheduling. *IEEE Embedded Systems Letters (ESL)*, 2013.

[104] S. Zhao. *A FIFO Spin-based Resource Control Framework for Symmetric Multiprocessing*. PhD thesis, University of York, 2018.

[105] S. Zhao, W. Chang, R. Wei, W. Liu, N. Guan, A. Burns, and A. J. Wellings. Priority assignment on partitioned multiprocessor systems with shared resources. *IEEE Transactions on Computers (TC)*, 2020.

[106] S. Zhao, N. Chen, Y. Fang, Z. Li, and W. Chang. A universal method for task allocation on fp-fps multiprocessor systems with spin locks. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2023.

[107] S. Zhao, X. Dai, and I. Bate. Dag scheduling and analysis on multi-core systems by modelling parallelism and dependency. *IEEE Transactions on Parallel and Distributed Systems*, 2022.

[108] 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.

[109] S. Zhao, J. Garrido, A. Burns, and A. Wellings. New schedulability analysis for MrsP. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10. IEEE, 2017.

[110] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys (CSUR)*, 45(1):1–28, 2012.