THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Three Aspects of Real-Time Multiprocessor Scheduling: Timeliness, Fault Tolerance, Mixed Criticality

RISAT MAHMUD PATHAN

Division of Networks and Systems
Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2012

Three Aspects of Real-Time Multiprocessor Scheduling: Timeliness, Fault Tolerance, Mixed Criticality
Risat Mahmud Pathan
Göteborg, Sweden, 2012
ISBN: 978-91-7385-754-3

Doktorsavhandlingar vid
Chalmers tekniska högskola
Ny serie Nr 3435
ISSN 0346-718X

Technical Report No. 86D
Dependable Real-Time Systems Group
Department of Computer Science and Engineering
Chalmers University of Technology

Contact Information:

Division of Networks and Systems
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96, Göteborg, Sweden
Phone: +46 (0)31-772 10 00
Fax: +46 (0)31-772 36 63
`http://www.chalmers.se/cse/`

Printed by Chalmers Reproservice
Göteborg, Sweden 2012

Three Aspects of Real-Time Multiprocessor Scheduling: Timeliness, Fault Tolerance, Mixed Criticality

Risat Mahmud Pathan
Department of Computer Science and Engineering
Chalmers University of Technology, Sweden

Abstract
The design of real-time systems faces two important challenges: incorporating more functions/services on existing hardware to make the system more attractive to the market, and deploying existing software on multiprocessors (e.g., multicore) to utilize more processing power. Adding more services on the same hardware needs efficient resource utilization. In addition, satisfying the real-time constraints, while at the same time efficiently utilizing the multiprocessor platform, is a challenging problem. This thesis deals with *global multiprocessor scheduling* for real-time systems, that is, the *fixed-priority* scheduling of sporadic tasks, where each task is allowed to run on any processor.

More specifically, this thesis considers *three* aspects of the design and analysis of global scheduling algorithms: timeliness, fault tolerance, and mixed criticality. *Timeliness* is about meeting the deadlines of the tasks; *fault tolerance* is about producing the correct output within the deadline even in the presence of faults; and *mixed criticality* is about facilitating the certification of systems when tasks having different criticality (or importance) are hosted on a common computing platform.

With respect to the timeliness aspect, global multiprocessor scheduling is analyzed (by assuming no faults and the same criticality for all the tasks) in order to propose new fixed-priority assignment policies and efficient schedulability tests. The proposed schedulability tests are shown to not only dominate (from a theoretical point of view) but also significantly outperform (by using simulation experiments) the state-of-the-art schedulability tests for global fixed-priority scheduling.

To allow for the combination of fault tolerance and timeliness, new scheduling algorithms that use time redundancy (i.e., execution of backup task) to tolerate multiple hardware and software faults are proposed. To account for the potential intrusive effect of time-redundant execution of backup tasks on the capability to meet task deadlines, new efficient schedulability tests for the proposed algorithms are derived. If a task set satisfies the schedulability tests, then all the task deadlines are met even when multiple faults (restricted by the assumed fault model) are to be tolerated using time redundancy.

To allow mixed-criticality tasks to be hosted on the same multiprocessor platform, a new algorithm for fixed-priority scheduling is proposed. The purpose of the algorithm is to facilitate certification, while at the same time efficiently utilizing the processing platform. A schedulability test for the algorithm can determine whether the appropriate level of assurance, according to the requirement of some certification authority/standard for meeting the deadlines of the mixed-criticality tasks, is guaranteed or not.

Keywords: Real-Time Systems, Sporadic Tasks, Fixed Priority, Global Multiprocessor Scheduling, Time Redundancy, Fault-Tolerant Scheduling, Mixed-Criticality Systems

List of Publications

This thesis is based on and extends the results in the following works:

- Risat Mahmud Pathan, "Schedulability Analysis of Mixed-Criticality Systems on Multiprocessors," *24th Euromicro Conference on Real-Time Systems (ECRTS), Pisa, Italy, 2012.*

- Risat Mahmud Pathan and Jan Jonsson, "A New Fixed-Priority Assignment Algorithm for Global Multiprocessor Scheduling," *Technical Report No. 2012:10, Department of Computer Science and Engineering, Chalmers University of Technology, Sweden, 2012.*

- Risat Mahmud Pathan and Jan Jonsson, "FTGS: Fault-Tolerant Fixed-Priority Scheduling on Multiprocessors," *8th IEEE International Conference on Embedded Software and Systems (ICESS), Changsha, China, 2011.*

- Risat Mahmud Pathan and Jan Jonsson, "Improved Schedulability Tests for Global Fixed-Priority Scheduling," *23rd Euromicro Conference on Real-Time Systems (ECRTS), Porto, Portugal, 2011.*

- Risat Mahmud Pathan and Jan Jonsson, "Exact Fault-Tolerant Feasibility Analysis of Fixed-Priority Real-Time Tasks," *16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Macau SAR, P.R.C., 2010.*

The following works are related but not covered in this thesis:

- Risat Mahmud Pathan and Jan Jonsson, "Load Regulating Algorithm for Static-Priority Task Scheduling on Multiprocessors," *24th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Atlanta, USA, 2010.*

- Risat Mahmud Pathan and Jan Jonsson, "Parameterized Schedulability Analysis on Uniform Multiprocessors," *39th International Conference on Parallel Processing (ICPP), San Diego, CA, USA, 2010.*

- Risat Mahmud Pathan, "Fault-Tolerant Real-Time Scheduling using Chip Multiprocessors," *Proceedings Supplemental volume of the 7th European Dependable Computing Conference (EDCC), Kaunas, Lithuania, 7-9 May, 2008.*

- Johan Nordlander, Rolf Johansson and Risat Mahmud Pathan, "Unambiguous Semantics In Automotive Timing Modeling," *1st Workshop on Critical Automotive applications: Robustness & Safety (CARS) in conjunction with the 8th European Dependable Computing Conference (EDCC), Valencia, Spain, 2010.*

Acknowledgments

First of all, I would like to thank my advisor Professor Jan Jonsson for his excellent comments, invaluable ideas, feedback, and most importantly, his confidence in me to carry out this research. It has been an extreme pleasure and a privilege working with and learning from him. I am grateful to him for supervising my work over the past couple of years with great patience and enthusiasm. I am thankful to Jan also for sponsoring my trips to several conferences, which have helped me to learn and to get better understanding of the real-time systems community.

Special thanks and gratitude to my thesis examiner Professor Johan Karlsson for his feedback and sharing with me his knowledge on fault-tolerant computer systems. I thank Professor Koen Claessen, Professor Per Stenström and Professor Philippas Tsigas for helpful discussion about the direction of research during my PhD follow-up meetings. Thanks to Dr. Johan Nordlander, Dr. Rolf Johansson and Dr. Anders Svensson who I worked with and learned from while working on the TIMMO and MCC-AI projects. Thanks to former gratuates Dr. Raul Barbosa and Dr. Daniel Skarin for their valuable suggestion and discussion during my early years as a PhD student. Thanks to Peter Lundin, head of my division, for helping me in dealing with administrative issues.

I am extremely grateful and would like to take the opportunity to thank Professor Sanjoy Baruah for hosting me as a visiting researcher in the Real-Time Systems Group at the University of North Carolina (UNC) at Chapel-Hill, USA in Fall, 2011. His guidance has always been a source of inspiration for research during my stay at UNC. I also thank all the members of the Real-Time Systems Group at UNC for their friendship.

Many thanks to all my colleagues at the Department of Computer Science and Engineering at Chalmers for creating such a friendly and stimulating working environment. Thanks to Associate Professor Roger Johansson and Arne Dahlberg for their help and advice regarding my role as teaching assistant in different courses at Chalmers. Thanks to Eva Axelsson, Peter Helander, Marianne Pleén-Schreiber, Tiina Rankanen, and other administrative personnel for helping me with different office-related matters. I am very thankful for the friendship that I have received from Alen, Angelos, Anurag, Behrooz, Bhabi, Dmitry, Erik, Fatemeh, Jakob, Kashab, Madhavan, Negin, Ruben, Tung, and all other PhD and post-doctoral students. Special thanks to former graduates Dr. Mafijul Islam and Dr. M.M. Waliullah who helped me settling — received me at the airport, cooked me dinner, helped me finding an apartment — when I first arrived in Sweden.

Thanks to the anonymous reviewers in the research community who reviewed our submitted manuscripts and gave comments in improving our work before publication.

I would like to thank the Swedish Agency for Innovation Systems (VINNOVA) for funding this research under the TIMMO (P30619-2), TIMMO-2-USE (39005), NFFP-4 (S4207), and Multi-Core Computing in Automotive Industry (MCC-AI) projects.

I want to express my deepest gratitude and thanks to my parents, brother and sister who have been always encouraging me in pursuing my study. Without their support and care I would not have finished doing this work. Finally, I thank my wife Nashita Moona and our son Mahir Samran Pathan for their love, continuous support, and particularly, for their patience during the last five years of my PhD study. Thank You!

Risat Mahmud Pathan
October 16, 2012

Contents

Abstract iii

List of Publications v

Acknowledgments vii

1 Introduction 1
- 1.1 Context of this Research . . . 3
- 1.2 Contribution Areas . . . 7
  - 1.2.1 Timeliness . . . 7
  - 1.2.2 Timeliness vs. Fault-Tolerance . . . 10
  - 1.2.3 Timeliness vs. Mixed-Criticality . . . 14
- 1.3 Applicability of this Research . . . 16

2 Preliminaries 19
- 2.1 Real-Time Systems . . . 19
  - 2.1.1 Sporadic Task Systems . . . 19
  - 2.1.2 Task Priority . . . 21
  - 2.1.3 Preemptive Scheduling . . . 21
  - 2.1.4 Work-Conserving Scheduling . . . 21
  - 2.1.5 Schedulability and Optimality . . . 22
  - 2.1.6 Schedulability Test . . . 22
  - 2.1.7 Minimum Achievable Density . . . 23
  - 2.1.8 Scheduling Algorithms . . . 24
- 2.2 Fault-Tolerant Systems . . . 27
  - 2.2.1 Failure, Error, and Fault . . . 27
  - 2.2.2 Error Detection Techniques . . . 29
- 2.3 Mixed-Criticality Systems . . . 30

3 Models 33
- 3.1 Task Model . . . 33
- 3.2 Resource Model . . . 37
- 3.3 Fault Model . . . 38

4 Goals and Contributions **41**

5 Density-Bound-Based Test **45**
5.1 Introduction . . . 46
5.2 Related Work . . . 48
5.3 Parameters of Task Model . . . 49
5.4 Constrained-Deadline Tasks: Density-Bound . . . 50
5.4.1 Prior Results and Useful Definitions . . . 51
5.4.2 "Special" Task Set and its Schedulability . . . 53
5.4.3 Slack-Monotonic Hybrid Priority Assignment . . . 56
5.4.4 Density Bound for Policy ISM-DS . . . 58
5.5 Policy ISM-DS[ξ]: Searching the Threshold . . . 62
5.6 Empirical Investigation . . . 65
5.6.1 Task Sets Generation Algorithm . . . 66
5.6.2 Result Analysis . . . 67
5.7 Implicit-Deadline Tasks: Utilization Bound . . . 72
5.7.1 Independent and Scale Invariant Priority Assignment . . . 73
5.8 Uniprocessor Slack-Monotonic Scheduling . . . 75
5.9 Summary . . . 76

6 Iterative Tests **77**
6.1 Introduction . . . 78
6.2 An Analysis Framework . . . 81
6.2.1 Audsley's OPA Algorithm . . . 82
6.3 Related Work . . . 85
6.3.1 State-of-the-art Iterative Tests . . . 86
6.4 The H-ODA-LC Test . . . 89
6.4.1 Applying HPA Policy to ODA-LC Test . . . 89
6.5 The IA-DA Test . . . 92
6.5.1 Overview of the IA-DA Test . . . 92
6.5.2 New Criterion for Separation . . . 93
6.5.3 Priority Assignment Algorithm: the IA-DA Test . . . 99
6.6 The IA-RT Test . . . 103
6.6.1 The D-RTA-LC Test . . . 103
6.6.2 Priority Assignment Algorithm: the IA-RT Test . . . 104
6.7 Empirical Investigation . . . 107
6.7.1 Result Analysis . . . 108
6.8 Summary . . . 112

7 Fault-Tolerant Scheduling on Uniprocessor **113**
7.1 Introduction . . . 113
7.2 System Model . . . 115
7.2.1 Traditional Deadline-Monotonic (DM) Scheduling . . . 117
7.3 Related Work . . . 117
7.4 Problem Formulation . . . 119
7.5 Load Factors and Composability . . . 121
7.5.1 Calculation of `Load-Factor-i` . . . 122
7.5.2 Calculation of `Load-Factor-HPi` . . . 124
7.6 Exact Schedulability Test . . . 138
7.7 Algorithm for the `FTDM` Schedulability Test . . . 141
7.7.1 Multiprocessor Scheduling . . . 144
7.8 Summary . . . 145

8 Fault-Tolerant Scheduling on Multiprocessors **147**
8.1 Introduction . . . 147
8.2 Related Work . . . 149
8.3 System Models and the `FTGS` Scheduling . . . 151
8.4 Problem Statement . . . 152
8.5 Analysis for Tolerating Task Errors . . . 153
8.6 Calculating Interfering Workload . . . 155
8.6.1 Workload of task τ_i . . . 156
8.6.2 Interfering Workload of task τ_i . . . 160
8.7 Total Interfering Workload of the Tasks in $\mathtt{HP}_k$ . . . 162
8.7.1 Finding Carry-in Set $\mathcal{Q}(S, a, \hat{m}, c)$ . . . 162
8.7.2 Total Interfering Workload and Schedulability Test . . . 165
8.8 Tolerating Processor Failures . . . 169
8.9 Graceful Degradation . . . 171
8.9.1 Direct Rejection . . . 172
8.9.2 Criticality-Based Eviction . . . 172
8.9.3 Imprecise Computation . . . 173
8.10 Summary . . . 173

9 Mixed-Criticality Systems **175**
9.1 Introduction . . . 175
9.2 System Model and The Scheduler . . . 178
9.3 Schedulability Analysis: an Overview . . . 180
9.3.1 Dual-Criticality Systems . . . 181
9.4 RTA Procedure at `LO` Criticality Level . . . 182
9.4.1 New RTA for Sporadic Task Systems . . . 182
9.5 RTA Procedure at `HI` Criticality Level . . . 184
9.5.1 Workload of $\tau_k \in$ `hpL(i)` within $[r_i^x, r_i^x + t)$ . . . 185

9.5.2 Workload of $\tau_k \in$ `hpH(i)` within $[r_i^x, r_i^x + t)$ 186
9.5.3 The RTA Test for `HI` Criticality Level 190
9.6 Schedulability Analysis for $\mathcal{L} > 2$ 192
9.6.1 Finding Priorities using Audsley's Algorithm 194
9.7 Empirical Investigation 195
9.8 Related Works 197
9.9 Summary 200

10 Conclusion **201**

A Proofs of Theorems and Lemmas **219**

B Additional Graphs for Iterative Tests **229**

1 Introduction

The demand for more functionalities and comfort in the use of today's prevailing computerized systems is increasing. The types and varieties of different functions or services determine the competitiveness of computerized systems — e.g., portable devices, cars, aircrafts — in the market. A modern passenger car, now-a-days equipped with dozens of processors, does not only provide functions related to vehicle control but also supports services related to comfort and safety. The development of such complex computerized systems with increasingly higher number of functionalities requires rigorous design and analysis effort to ensure that the system is "predictable".

In my opinion, a system is predictable if any possible run-time behavior and its consequences are either known or can be tuned to be known during the design of the system. One way to characterize a computerized system is based on its functional and non-functional behaviors. The *functional behaviors* of a system are the main activities or services provided by the system, for example, anti-lock braking system (ABS) in a car, online stock trading service, auto-pilot function in an aircraft, and so on. The end-users directly interact with the functional behaviors of a system. The *non-functional behaviors* are the qualitative or quantitative measure of the functional behaviors. Examples of non-functional behaviors of computerized systems are throughput, timeliness, and energy consumption (e.g., in portable devices). The users perceive the non-functional behaviors while interacting with the functional behaviors of the system.

The aim of the research presented in this thesis is to aid the system designer in ensuring predictability regarding some important non-functional behaviors of a class of computer systems known as *real-time systems*. The most prominent non-functional behavior of a real-time system is the requirement on producing the output within a certain deadline (also referred to as timeliness). Examples of such systems are automotive,

avionics, space systems, nuclear power plants, and consumer electronics. This thesis focuses on modeling, analysis, and verification of some important non-functional behaviors of real-time systems.

The modeling and analysis of key non-functional behaviors of real-time systems are important to ensure predictability. This is because the popularity and success of a computerized system does not only depend on *what* it does but also on *how* it does it. Consider the example of withdrawing money from an ATM where a customer enjoys the opportunity of getting cash in a remote location without visiting the bank in person. However, the client would not be satisfied if the ATM does not dispense the cash few seconds after correct credentials are entered into the machine's keypad. The ability to withdraw cash at a remote location is a functional behavior of the ATM while the time it takes in dispensing cash is an important non-functional behavior. Another non-functional behavior for an ATM system is its fault-tolerance capability: after withdrawing cash from an ATM, the account balance of the customer must not be updated incorrectly even if the system encounters some fault. Acceptable non-functional behaviors are crucial to customer satisfaction with the functional behaviors of the system.

What is modeling? In the context of this thesis *modeling* refers to the act of formally representing the parameters and assumptions of the system relevant to the non-functional behaviors under study. In particular, the software (e.g., functional behaviors) and hardware (e.g., number and type of processors) are abstracted using modeling. In addition, the constraints needed for *acceptable* non-functional behavior of the system are formally captured. Modeling eliminates unnecessary details and captures only the relevant information necessary for analyzing the non-functional behavior under study. For example, dispensing cash within 5 seconds, after correct credentials are entered, may be an acceptable non-functional behavior to most of the clients of an ATM. This non-functional behavior, i.e., time it takes in dispensing cash, can be modeled using a parameter called "$dispenseTime$". And, the acceptable behavior of $dispenseTime$ can be modeled as a constraint such that "$dispenseTime \leq 5\ seconds$".

What is analysis? In the context of this thesis *analysis* refers to evaluating the non-functional behavior from the worst-case perspective. Analysis is about determining the worst-case situation that might occur at run-time and (qualitatively or quantitatively) evaluating the non-functional behavior during that particular situation. However, identifying the worst case may not be trivial or its analysis may not be simple or straightforward. In the ATM example, finding the worst case of the non-functional behavior $dispenseTime$ means finding the maximum time the ATM takes in dispensing cash. And, the analysis of $dispenseTime$ requires the consideration of several factors for example, hardware, software, network latency, time to check the customer's account balance, and so on. Determining the exact worst-case behavior of the system, considering a particular non-functional behavior, may not be always possible due to shortage of time, limited resources, or due to the complexity of the analysis. In such case, the worst-case behavior may need to be safely approximated which introduces pessimism in the analysis. The degree of pessimism determines the preciseness of analysis — lower pessimism (without compromising the correctness) means more precise analysis.

Is the system acceptable? Whether a non-functional behavior is acceptable or not is determined using *verification*. While analysis estimates the quality of the non-functional behavior from the worst-case viewpoint, verification is about checking whether this quality is acceptable to the customers or compliant with some certification standard. For example, if the analysis of non-functional behavior $dispenseTime$ concludes that $dispenseTime = 20\ seconds$, then the verification step would conclude that the behavior $dispenseTime$ is not acceptable because "$dispenseTime > 5\ seconds$". Unacceptable non-functional behaviors may require changes in hardware, software or even a re-specification of the system. Clearly, such changes cost significant time and/or money.

Appropriate modeling, effective analysis and efficient verification of non-functional behaviors of real-time systems are therefore of utmost importance and are also the main ingredients of this thesis.

1.1 Context of this Research

The non-functional behaviors of a system may not be specified by the end-users, e.g., buyers of passenger cars.[1] The end-users may remain unaware of the important non-functional behaviors of the system. However, the end-users become aware of the existence and importance of a non-functional behavior if its quality becomes unsatisfactory in some way. The level of acceptability of a particular non-functional behavior is modeled as one or more *design constraints* which are verified before the system is put in mission. It is the responsibility of the system designers to ensure that functional behaviors are correctly implemented and that the design constraints used to model the acceptability of the non-functional behaviors are satisfied. This thesis addresses the modeling, analysis, and verification of *three* important non-functional behaviors of real-time systems: *timeliness, fault tolerance,* and *mixed criticality*.

Timeliness is a non-functional behavior which is about meeting the deadlines of the real-time applications deployed on a particular computing platform. Acceptable timeliness behaviors of real-time application are specified as timing constraints. The *first* research question addressed in this thesis considers timeliness:

> **Q1** *How to guarantee that all the deadlines of a real-time application are met on a particular computing platform?*

Fault tolerance is a non-functional behavior which is about providing correct service even in the presence of faults. Fault-tolerant behavior is implemented using hardware (space) or software (time) redundancy in many safety-critical systems, for example, automotive, aircraft, and space shuttle applications. This thesis considers fault-tolerant

[1] However, the OEM (not an end-user) of a passenger car may specify the non-functional behaviors when ordering or buying particular component from an external supplier. Non-functional behaviors of defense applications are often specified by the corresponding military organization.

systems that are also real-time systems. Deviation from acceptable timeliness or fault-tolerant behavior of such systems might result in catastrophic consequences, for example, loss of human lives, threat to the environment or severe economic loss.

The timeliness and fault-tolerant behaviors may be dependent on one another in a conflicting way. For example, the likelihood of meeting timing constraints of a fault-tolerant system may decrease as the amount of space or time redundancy used to achieve fault-tolerance is increased. In other words, the requirement on timeliness in such case is competing with the requirement on fault-tolerant behavior. To that end, the *second* research question addressed in this thesis considers this interdependency of timeliness and fault-tolerance:

> **Q2** *How to guarantee that all the deadlines of a real-time application are met on a particular computing platform while providing fault tolerance using time or space redundancy?*

Mixed criticality is a non-functional behavior which is about providing certain level of assurance regarding the correct behavior (e.g., meeting the deadlines) of different multi-criticality functions hosted on a common computing platform. Traditionally, the design of a non-mixed-criticality system assumes the *same* criticality level for all the functions present in the system. In contrast, an Mixed Criticality (MC) system has *multiple* criticality levels where each function is assigned one unique criticality based on its "importance". For example, the ABS function in a car is assigned a safety criticality level that is relatively higher than that is assigned to the DVD player function. Higher criticality level assigned to a function means that higher degree of assurance is needed regarding the correct behavior of the function.

The design of safety-critical systems considers the integration of multiple functions having different criticality levels on a single, powerful processor due to space, weight and power (SWaP) concerns. The run-time behavior of such systems varies based on the operating environment, hardware dynamics, input parameters, and so on. The behavior of the system at each time instant determines the *criticality behavior* of the system at that time instant. The criticality behavior of the system changes from one time instant to another while the statically-assigned criticality of each function does not change.

MC systems often need to be certified by a third party, known as a certification authority (CA). Certification is about ensuring certain level of confidence regarding the acceptable (i.e., correct) behavior of the system. For example, certifying an aircraft may need to verify that standard design guidelines are followed during the development of the flight-control software. A certified product is considered safe and also promotes confidence among the end-users in buying that product. The degree of assurance needed for certifying the behavior of an MC system as "correct" at one criticality level is typically different from the assurance needed at a different criticality level. In this thesis, the correct behavior of an MC system is modeled using timing constraints (i.e., deadlines).

Whether the deadline of a function is met or not depends on the worst-case execution time (WCET) of the function, which is the maximum CPU time the function requires to complete its execution. The WCET of a function can be approximated at varying degrees

of confidence or assurance, depending on the inaccuracy or difficulty in estimating the true WCET, for example, due to the variability in inputs, operating environment, hardware dynamics, and so on. The higher degree of assurance needed in estimating the WCET of a function, the larger (more conservative) the WCET bound tends to be in practice. The criticality behavior of the system is then determined by comparing the actual execution time of each function with the WCET that is estimated using different degrees of assurance.

Conventional real-time scheduling policies for non-MC systems can not address both deadline and criticality (e.g., multiple WCET of the same function). The *third* research question addressed in this thesis considers this interdependency between timeliness and mixed-criticality:

> **Q3** *How to guarantee that all the deadlines of a real-time application are met while ensuring certification at each criticality level?*

In the context of this thesis, timeliness is about meeting deadlines; fault tolerance is about providing correct service even in the presence of faults while satisfying the timing constraints; and mixed criticality is about certifying the integration of mixed-criticality functions considering varying degrees of confidence in the WCET estimation of each function. The first problem considers the timeliness requirement independent of other non-functional behaviors while the second and third problems address the interdependency of different non-functional behaviors: timeliness vs. fault-tolerance and timeliness vs. mixed-criticality, respectively.

Application Characteristics. Many real-time applications, e.g., control and monitoring, are modeled as a collection of recurrent tasks with stringent/hard deadlines. A task is a particular piece of program code that performs some computation, e.g., reading sensor data, writing actuator value, executing a control loop, etc. The recurrent task model considered in this thesis is the *sporadic task model* where the inter-arrival time (period) of each task has a lower bound and the relative deadline of each task is not greater than its period. An instance (also, called job) of the task is said to be released when it becomes available for execution. The releases of two consecutive jobs are separated by at least the period of the task. The deadline is "relative" in the sense that whenever a job is released, the deadline for that job applies with respect to its release time. Each task is also characterized by exactly one WCET (i.e., the maximum CPU time the task requires to finish its execution[2]). Every job of the task must finish its execution before its corresponding deadline expires (i.e., the timing constraint of the task).

The category of real-time systems having stringent timing constraints is called *hard* real-time systems. If the timing constraints of a hard real-time system are not satisfied, then the consequences may be catastrophic, for instance, threat to human lives.

[2]The non-functional behavior timeliness, when considered independent of other non-functional behaviors, is based on the modeling of non-MC systems. So, only one WCET of each task is considered. Different WCET of the same task is considered when modeling MC systems.

Consequently, it is of utmost importance for designers of hard real-time systems to ensure a priori that all the timing constraints will be met when the system is in mission. The timing constraints of hard real-time applications can be fulfilled using appropriate scheduling of the tasks on a particular hardware platform. *Scheduling* is the policy of allocating resources (e.g., CPU time, communication bandwidth) to the tasks of the application that are competing for the same resource. **Scheduling algorithms and their analysis that can be used to verify the timing constraints of hard real-time systems are at the heart of the research presented in this thesis.**

Computing Platform. The emerging Chip-Multiprocessors (CMPs) technology, where multiple processing cores are placed on the same chip, is attractive for real-time systems design due to the computation power provided by such technology. Major processor-chip manufactures have already shifted towards multicore architecture to overcome the heat and thermal limitations in the design of single-core processors. Multicore processors are commonplace in both general purpose (e.g., Intel's dual-, quad-core processors) and embedded domains (e.g., ARM's Cortex family of processors). The trend is now incorporating more and more cores on the same chip. Intel's Teraflop research chip has announced the design of an 80 core platform. The current shift towards multicores by prominent chip vendors indicates that the commercially available off-the-shelf processors in near future would be only multicores. To this end, this thesis considers real-time scheduling on a computing platform having multiple identical processors/cores.

Scheduling Policy. The dominating scheduling approach in industry for meeting the hard deadlines of application tasks is the fixed-priority (FP) scheduling policy, due to its flexibility, ease of debugging, and predictability. Under the FP scheduling strategy, each task is assigned a priority that never changes during the execution of the task. This thesis addresses preemptive *global* FP *scheduling* of sporadic tasks on a platform consists of identical processors or cores. In preemptive global FP scheduling, at each time instant, the highest-priority runnable[3] task is dispatched for execution if a processor is idle. If all the processors are busy and a relatively lower-priority task is executing on some processor, then the highest-priority runnable task is dispatched for execution on that processor by preempting the lower-priority task. The preempted task may later resume its execution on any processor (i.e., the assumed execution model allows migration).

Given the trend of widespread diffusion of multicore platform for real-time systems, there are several challenges in global scheduling on multiprocessors. It has already been shown by the researchers in the real-time systems community that the relatively mature theories and techniques applicable to analyze timeliness on uniprocessor platform are not applicable (i.e., perform poorly) to global multiprocessor scheduling. For example, while the best fixed-priority ordering of sporadic tasks is known for uniprocessor FP scheduling, the best priority ordering for global FP scheduling is not currently known. In addition, when the non-functional behavior timeliness is considered in addition to other non-functional behaviors like fault tolerance or mixed criticality, the schedulability analysis becomes even more difficult. This thesis addresses such

[3] A task is runnable if it has been released but has not completed its execution

challenges by proposing new techniques to analyze global multiprocessor scheduling in order to answer the three research questions mentioned above.

Why global multiprocessor scheduling? There are two main paradigms for multiprocessor FP scheduling of real-time tasks: the global approach and the partitioned approach. In the partitioned approach, each of the tasks is preassigned to exactly one processor and allowed to execute only on that processor (i.e., no migration is allowed). Each processor can execute the assigned tasks using some uniprocessor FP scheduling algorithm, for example, Deadline-Monotonic (DM) scheduling in which task with shorter relative deadline is given higher fixed priority. In the real-time research community, there is no clear evidence that one scheduling paradigm is superior to another: one task set that is deemed schedulable using global FP scheduling may be not schedulable using partitioned FP scheduling, and conversely. However, global scheduling is advocated in this thesis for several reasons. First, the open research problems related to global FP scheduling are very challenging. Second, the adoption of global scheduling in actual multicore systems is becoming more likely as various mechanisms (e.g., locked cache) are being proposed to reduce migration overhead. Third, global scheduling does not require an a priori assignment of tasks to the processors (finding an optimal task assignment to processors is known as an NP-hard problem) and provides the flexibility to execute a task on any processor by allowing migrations. Finally, global scheduling does not require reassignment of the tasks if a new task has to be accepted in the system, for example, due to function or component upgrade (such reassignment is needed for partitioned scheduling when tasks are presorted prior to assigning them to processors).

1.2 Contribution Areas

What follows in this section are the major challenges and the contributions in dealing with each of the research questions mentioned above.

1.2.1 Timeliness

The most important non-functional behavior of a *real-time system* is timeliness. In this thesis, timeliness means *meeting the deadlines* of a set of real-time sporadic tasks. The output of a task corresponds to the functional behavior while the time at which the output is generated is related to the non-functional behavior timeliness. The deadline by which the output has to be generated is modeled as a *timing constraint*.

The means to satisfy the timing constraints is to appropriately schedule the tasks on the processors. Whereas the uniprocessor real-time scheduling theory is considered very mature, a comprehensive multiprocessor scheduling theory has yet to be developed. Many of the well-understood traditional uniprocessor scheduling algorithms perform poorly (in terms of hard real-time schedulability) on multiprocessors. There is consequently a need for the design and analysis of multiprocessor scheduling algorithms. This thesis considers global FP scheduling and its analysis to verify whether all the deadlines of the tasks are met or not.

Research Challenges. The two major research challenges for global FP scheduling are: (i) *priority assignment problem*, and (ii) *schedulability testing problem*. In global FP scheduling of sporadic tasks, whether a particular task, say task τ, meets its deadline or not depends on the tasks having priorities higher than that of task τ. This is because the set of higher priority tasks determine the length of the cumulative time interval during which all the processors are busy executing these higher priority tasks while the task τ is awaiting execution (called the *interference* on task τ due to the higher priority tasks). Since the priority ordering of the tasks determines the set of tasks having higher priorities than the priority of each task, the interference that a particular task suffers depends entirely on the priority ordering. Therefore, deriving a good fixed-priority assignment policy for global FP scheduling is important to guarantee the schedulability of each task. A priority assignment is said to be *optimal* if given some priority ordering for which all the deadlines of the tasks are met, then the optimal priority assignment also guarantees the same. While the optimal fixed-priority ordering of sporadic tasks scheduled preemptively on a uniprocessor is known[4], the optimal fixed-priority ordering for preemptive global multiprocessor scheduling is still unknown.

Whether the deadlines of the hard real-time tasks are met or not needs to be determined offline based on a schedulability test. A schedulability test of a scheduling algorithm is a condition that, when satisfied for a given task set, guarantees that all the deadlines of the tasks are met using that scheduling algorithm. Deriving a schedulability test involves analyzing the worst-case behavior of the scheduling algorithm. The worst-case behavior for global FP scheduling of sporadic tasks is difficult to determine.[5] To circumvent this problem, the worst-case behavior of global FP scheduling algorithm is approximated by introducing some degree of pessimism during the schedulability analysis. The challenge is to introduce as little pessimism as possible during the analysis in order to derive a more effective schedulability test based on a more precise analysis.

Contributions. In order to address the two problems just discussed, new fixed-priority assignment policies and effective schedulability tests for global FP scheduling of sporadic tasks are proposed in this thesis. Two different flavors of global FP schedulability tests are proposed: *density-bound tests* and *iterative tests*. One of the most simple schedulability tests is the density-bound test in which it is only required to check exactly one condition: if the total density[6] of a sporadic task set is not greater than a threshold (called the *density bound*), then all the tasks meet their deadlines. A larger density bound means a better schedulability test. Moreover, the density-bound test relates the sum of the densities of all the tasks in a task set to the total available processing capac-

[4]Deadline-monotonic priority ordering is optimal for uniprocessor FP scheduling where each task's relative deadline is less than or equal to its period.

[5]The worst-case scenario in analyzing a FP scheduling algorithm is called a critical-instant (formally defined later). While the critical instant for uniprocessor FP scheduling is known, the critical instant for global FP scheduling is not known.

[6]The *density* of a task is the execution time required per unit of time within the relative deadline of the task. The total density of a task set is the sum of densities of all the tasks in that task set. The *utilization* of a task is the execution time required per unit of time within the period of the task. Formal definitions of these concepts will be presented shortly.

ity. Consequently, a density-bound-based test can be used not only to verify the timing constraints for some given processing capacity but can also be used to determine the sufficient processing capacity needed for satisfying a given set of timing constraints.

A new fixed-priority assignment policy, called *Improved Slack-Monotonic Density Separation* (ISM-DS), is proposed in this thesis and the corresponding density bound for global FP scheduling is derived. This thesis will show that the proposed density-bound-based test dominates the state-of-the-art density-bound test for global FP scheduling of sporadic tasks where the relative deadline of each task is not greater than its period. By domination it means that there are schedulable task sets that satisfy the proposed density-bound test for ISM-DS but do not satisfy the state-of-the-art density-based test, and that the converse does not apply. The density-bound test becomes the utilization-bound test when the relative deadline of each task is equal to its period.

Unlike the density-bound test, an iterative schedulability test requires one condition to be tested for each task: if the schedulability condition is satisfied for each task (checked iteratively), then the entire task set is schedulable. In this thesis, a new iterative test, called *Interference-Aware Response-Time* (IA-RT) test, is proposed. The derivation of this iterative test is based on reducing different sources of pessimism identified in the state-of-the-art schedulability analysis of global FP scheduling. As shown in this thesis, the IA-RT test dominates the state-of-the-art iterative test for global FP scheduling. In addition, empirical investigation using randomly generated task sets shows that the IA-RT test significantly outperforms the state-of-the-art iterative test.

Determining the fixed-priority assignment of the tasks for global FP scheduling is a challenging problem and the optimal priority ordering in such case is not known. An important property of the IA-RT test is that it checks the schedulability of each task while assigning the fixed priorities to the tasks. If all the tasks are assigned priorities based on the IA-RT test, then it is also true that the task set is schedulable using global FP scheduling according to the assigned priorities. This is a very important property since determining the fixed-priority assignment of the tasks for global FP scheduling is a challenging problem and the optimal priority ordering in such case is not known. Notice that this result does not imply that priority ordering found using the IA-RT test is also the optimal priority ordering for global FP scheduling. Optimality can only be claimed with respect to the IA-RT test.

Apart from being able to verify the timing constraints, the proposed density-bound-based and iterative schedulability tests for global FP scheduling approximate the worst-case behavior by reducing the pessimism in comparison to that present in the state-of-the-art iterative schedulability tests. Reducing such pessimism has several advantages. First, it reduces the demand on computing resources which in turn reduces the cost of the system for mass production. Second, lower computing resource means less space, weight and power consumption which are desirable in many resource-constrained embedded systems. Finally, efficient use of system resources enables incorporating more functionalities on the same computing platform without buying additional hardware. All these advantages provide better competitiveness of a product in the market.

1.2.2 Timeliness vs. Fault-Tolerance

Real-time systems with fault-tolerance requirements must provide correct service even in the presence of faults. In addition to satisfying the timing constraints, the functional correctness of the application must be guaranteed; otherwise, the consequence may be disastrous. For example, after the computer system failed in the London Stock Exchange on September 8, 2008, the stock trading halted for several hours; upsetting clients who trade an average $17.5 billion a day. The cause of such incorrect behavior of computer system is the occurrences of faults in the system. Both permanent and transient faults in hardware may occur due to, for example, hardware defects, electromagnetic interferences, or cosmic ray radiation. In addition, software faults (bugs) may remain undetected even after months of software testing and debugging.

A system *failure* occurs when a system deviates from the correct specified service. Such deviation from correct service is due to some incorrect state in the system which is called an *error*, i.e., an error is liable for a failure. The source or cause of an error is a *fault*. To better understand these concepts, consider the following example.

Example 1.1 (Faults, Errors, Failures). *Consider a safety-critical system that must invoke a function, called $action()$, to avoid catastrophic consequence (failure) if the temperature of the system's environment, measured using a temperature sensor, is 0^o Celsius(C). Assume that the sensor only reads temperature in units of Fahrenheit(F). The conversion rule $C = (F - 32) * 1.8$ can be used to convert F to C. Therefore, when the sensor reading is 32^oF, which is equivalent to 0^oC, then $action()$ must be invoked to avoid system failure.*

Algorithm `some_control_function()`

// The system fails if $action()$ is not invoked when temperature is 0^oC
1. $F \leftarrow$ <read from temperature sensor>
2. $C = (F - 3.2) * 1.8$; // instead of 32 in the rule, 3.2 is used (a fault)
3. **If** $C == 0$ **Then**
4. $action()$;
5. **End If**

Figure 1.1: *A simple program to understand fault, error, and failure*

*This service of the system is implemented in Figure 1.1 where the constant 3.2 in line 2 is mistakenly used instead of constant 32 for the conversion rule. Coding the rule using constant 3.2 is an example of a **fault**. This fault causes incorrect computation of C in line 2 (an incorrect state of the system), which is an **error**. When the read (input) value from the sensor is $F = 32$, the converted value $C = (32 - 3.2) * 1.8 = 51.84$ in line 2 is erroneous which results in a system **failure** because $action()$ in line 4 is not invoked although the actual temperature of the environment is $32^oF = 0^oC$.*

Not every error leads to a system failure. When the input from the sensor is not 32^oF (i.e., actual temperature of the environment is not 0^oC), the converted value in line 2 is erroneous; however, the system does not fail because function $action()$ is not needed to be invoked in such case anyway.

Similarly, every fault does not cause an error. To see why, consider that the sensor is not working properly and reads 3.2^oF when the "true" temperature of the environment is 32^oF. Although the conversion rule in line 2 is faulty, the state of the system is not incorrect (no error) since the converted value 0^0C is correct for the actual temperature of 32^oF. In such case, the fault in the conversion rule is masked, there is no error, and function $action()$ is invoked. □

The faults that are manifested as errors must be tolerated to prevent system failures without its effect being adversely perceived by the end-users (an acceptable non-functional behavior). However, no fault-tolerant system can tolerate an infinite numbers and arbitrary types of faults. The nature and frequency of faults considered for the design of a particular fault-tolerant system are specified using a *fault model*. The fault model used for analyzing the predictability of different computer systems varies. For example, the fault model considered during the design of a space shuttle is different from that of personal computers.

The level of protection needed against failures is modeled as *reliability constraints*. For example, the reliability constraint for the design of a fault-tolerant system may be to withstand a total of f transient errors (as caused by hardware transient faults). Satisfying the reliability constraints ensures that the functional behavior of the system is acceptable even in the presence of faults. Acceptable timeliness and fault-tolerance behaviors can be achieved by means of *fault-tolerant scheduling*, which is the focus of this thesis.

Research Challenges. Achieving fault-tolerance in computer systems requires employing redundancy either in space or in time. Space redundancy is provided by additional hardware, for example, using extra processors. Space redundancy is used to achieve tolerance against permanent hardware failure. For example, when a processor chip ceases functioning, the tasks can be executed on redundant processors. However, due to cost, volume and weight considerations implementing space redundancy for all the functionalities may not be always viable, for example, in space, automotive or avionics applications. To achieve fault-tolerance in such systems, time redundancy is used in the form of executing *backup* tasks.

Fault-tolerance using time redundancy in real-time systems can not be addressed independently of the task-scheduling issues. This is because time-redundant execution as a means for tolerating faults may have a negative impact on the schedule of the tasks in the sense that it might lead to missed deadlines for one or more of the tasks. Consequently, there is a need for fault-tolerant scheduling algorithms that minimize such intrusive impact resulting from time-redundant execution to tolerate faults.

Contributions. This thesis presents fault-tolerant FP scheduling algorithms for both uni- and multiprocessor platforms considering a certain fault model. The proposed fault model is very powerful in the sense that multiple faults can occur in any task and at any time, even during the execution of a recovery operation. Transient hardware faults that cause transient task errors is considered in the fault model. Transient hardware faults are short lived and generally cause no permanent error to the hardware. Therefore, re-executing the original task as backup is a cost-efficient and simple means for tolerating

such faults. Although software faults are permanent, their manifestation (i.e., the corresponding error) might be of transient nature due to, for example, changes in input or executing a different path during re-execution. Such software faults which result in transient errors are considered in the fault model and can be tolerated through re-execution. Software faults, which result in permanent task errors (and therefore can not be tolerated using re-execution), are also considered in the fault model. A diverse implementation of the task needs to be executed as backup to recover from such permanent error due to software faults. A diverse implementation of the same task is expected not to have the same software fault, and therefore, does not cause the same permanent error upon execution of the backup.

The types of faults considered in the fault model can cause task error (i.e., incorrect output generated by the task) or permanent processor failure (i.e., some processors in the multiprocessor platform are not working). The concepts of task error and processor failure are distinguished in this thesis. A *task error* corresponds to a situation where the output of a task is not correct but the processor on which the task is executing is non-faulty. A *processor failure* is caused by a fault that is permanent in hardware and the output of the task executing on that faulty processor is considered as erroneous. Mitigating the effect of a processor failure does not mean that the failed processor becomes functional again; instead, it means the task that was executing on the failed processor still meets its deadline by executing its backup on a non-faulty processor. The fault-tolerant scheduling algorithms proposed in this thesis consider original-task re-execution or diverse-implementation execution as backup in order to tolerate both task errors and processor failures. Any instance of a task when first executes is called the *primary* while the original-task re-execution or diverse-implementation execution is called the *backup*.

This thesis proposes a preemptive FP uniprocessor fault-tolerant algorithm, called Fault-Tolerant Deadline-Monotonic (FTDM) scheduling, where at most f task errors can be tolerated within all possible time intervals, each of which is not longer than the maximum relative deadline of any task in the task set. The FTDM algorithm is designed not to consider permanent processor failure in its fault model.[7] The FTDM scheduling considers tolerating task errors caused by hardware or software faults. The errors affecting the tasks are tolerated using time redundancy where both original-task re-execution or diverse-implementation execution as backup is possible. When an error is detected, the backup of the task becomes ready for execution. An exact schedulability condition of the FTDM algorithm is derived, for which it applies that all task deadlines are met if, and only if, this condition is satisfied.

While processor chip failures requires that redundant chips are available to tolerate permanent hardware failures, permanent core failure in CMPs may be tolerated without having additional backup processor chip. The task that was executing on a faulty core can be migrated to a non-faulty core on the same chip and its backup can be executed on this non-faulty core. Such time-redundant execution on the same chip is possible if the task set is still schedulable on the remaining available (non-faulty) processing

[7]However, the exact schedulability condition for the FTDM algorithm is directly applicable to partitioned multiprocessor scheduling where tasks are preassigned to processors and never migrate.

cores. Luckily, contemporary CMPs offer such high processing capacity that they may be exploited to tolerate core failures. Therefore, time redundancy in combination with space redundancy can mitigate the effect of permanent core failures in CMPs where the scheduling algorithm allows task migration.

Most of the previous work on fault-tolerant scheduling for multiprocessors, based on partitioned method, do not distinguish between tolerating task error and processor failure. Previous work considered tolerating task error by pessimistically assuming that the processor on which the faulty task was executing has crashed and execute the backup task on a different processor to which the backup is preassigned. Such pessimism unnecessarily increases the number of processors required to tolerate task errors even though it could be possible to execute the backup on the same (non-faulty) processor on which the task error is detected. Moreover, increasing the number of processors is costly in terms of SWaP for many embedded real-time systems and also increases the probability of having more faults as more chips are deployed.

To this end, this thesis proposes a multiprocessor FP fault-tolerant scheduling algorithm, called Fault-Tolerant Global Scheduling (FTGS), which tolerates both task errors and processor failures. The design of the FTGS algorithm is based on two crucial observations: (i) in case of task error, the global scheduler can simply dispatch the backup of a faulty task to any processor (even to the processor on which the task encountered the error), and (ii) mitigating the effect of processor failure is same as tolerating a task error by dispatching the backup of the task to a non-faulty processor. The FTGS algorithm considers tolerating f task errors within all possible intervals not larger than the maximum relative deadline of any task and tolerates (i.e., mitigate the effect of) total ρ permanent processor failures during the entire lifetime of the system.

The schedulability analysis for the FTGS algorithm derives a schedulability test that, when satisfied, guarantees that all the deadlines of the tasks are met even in the presence of task errors and processor failures. The novelty of the proposed schedulability test is that the resilience in terms of tolerating different combinations of task errors and processor failures can be efficiently determined for resource-constrained embedded real-time systems. Moreover, if the given priority ordering of the task set does not satisfy the proposed test, then a priority ordering for which the task set may satisfy the proposed test can be searched efficiently. Finding such a priority ordering is important since it avoids unnecessary upgrading of the hardware or even re-specification of the software.

The proposed schedulability tests for the FTDM and FTGS scheduling algorithms can be used to verify the reliability and timing constraints (under the assumed task and fault models) for uni- and multiprocessors, respectively. The mathematical expressions of these schedulability tests include parameters related to the task, fault and resource models. The system designer can play around with these parameters to determine, for example, the resource requirement for a given task set and fault model, or the maximum number of task errors that can be tolerated on a given processing platform. Such capability enables the designer to make a trade-off between resource requirement and the level of redundancy necessary for an acceptable fault-tolerant behavior of the system.

1.2.3 Timeliness vs. Mixed-Criticality

SWaP concerns drive the design of safety-critical systems towards integrating multiple functionalities having multiple criticality levels on the same computing platform. The computation power of CMPs also encourages such integration so as to incorporate more functionalities on the same platform. For example, aviation industry is contemplating "Integrated Modular Avionics" (IMA) to achieve economic advantage by hosting multiple avionics functions on a single platform.

Traditionally real-time scheduling of safety-critical systems assumes that all the tasks in the system have the same level of criticality (or importance). In contrast, an MC system is one in which the *criticality levels* of different real-time tasks may be different. For example, in the RTCA DO-178B standard, there are five different Design Assurance Levels (DAL A to DAL E) for software in avionics systems, and in ISO 26262 standard, safety functions in automotive systems can have four different Automotive Safety Integrity Levels (ASIL A to ASIL D). In this thesis, it is assumed that assigning a criticality level to a task in an MC system means the degree of assurance required for the correct behavior (i.e., meeting deadline) of that task.

In order to certify an MC system as being correct, the CAs make certain assumptions about the worst-case run-time behavior of the system. However, the assumptions made by the system designers may be different from that of the CAs. For example, the CA generally makes very conservative assumptions regarding the WCET of a task in comparison to the assumptions made by the system designers. It is well-known that the accuracy in estimating the WCET of a particular piece of code is problematic: the WCET used for the schedulability analysis of each task is generally a conservative upper bound that exceeds the true WCET. The higher level of assurance or confidence needed in estimating the WCET of a piece of code, the larger the WCET tends to be in practice.

Different upper bounds on WCET for a piece of code can be considered based on the level of assurance needed for certification at different criticality levels. Whether the deadline of a task is met depends on the WCET of the task, and therefore, the level of assurance needed for certification in meeting the deadlines depends on the level of assurance used in deriving the WCET of that task. When certifying the system at a lower criticality level than the criticality assigned to some task, the WCET of that task estimated according to the level of assurance required at that lower criticality level can be considered during the schedulability analysis of MC system.

Research Challenges. One of the challenges regarding the design of MC systems is to ensure the *isolation* property, i.e., that functions, tasks or components at a lower criticality level do not interfere adversely with those at a higher criticality level. Such level of isolations can be provided by dedicating the system resources for each criticality level. For example, all the high critical functions may be integrated on a separate processor. However, providing a dedicated resource at each criticality level may not be cost- and resource-efficient. Therefore, sharing the computing resources among the tasks having multiple criticality levels has to be considered. Unfortunately, such sharing requires special design considerations to avoid issues such as, the *criticality inversion problem*,

where a high critical task may miss its deadline when the scheduler assigns CPU time to meet the deadline of a low criticality task.

Although the isolation property is not explicitly addressed in this thesis, global FP scheduling can achieve this property as follows. When multiple tasks having different criticality levels are integrated on the same multicore chip, all the tasks having the same criticality can be globally FP scheduled on a (dedicated) subset of the processing cores. This scheduling approach requires no explicit task assignment algorithm, and more importantly, the temporal behavior of each function can be restricted only to its dedicated cores to ensure isolation. Such restriction is necessary and beneficial for function/component upgrade, modification and incremental certification.

Another challenge in the design of mixed-criticality scheduling is the priority assignment problem for the MC tasks. The priority and criticality of a task are not necessarily positively correlated in the sense that always assigning higher priority to a higher criticality task may not yield the best performance. The criticality level of a task is statically assigned based on the degree of assurance needed regarding its correct behavior (which in this thesis is about meeting its deadline). In case of FP scheduling, a task with higher criticality level may sometimes be assigned higher fixed priority to ensure, for example, the isolation property or to avoid the criticality inversion problem. However, a task with higher criticality level may sometimes need to be assigned a relatively lower fixed priority to allow the deadlines of all the tasks to be met. Assigning higher fixed priority to higher criticality task is known as the Criticality-As-Priority-Assignment (CAPA) policy. It will be evident later that CAPA is not an optimal priority assignment policy for FP scheduling of MC tasks. In fact, the optimal FP ordering of MC tasks is still unknown for multiprocessors. Therefore, determining a good FP priority assignment policy is very important for MC systems and this problem is addressed in this thesis.

A third aspect in the design of MC systems is *static verification*, which is related to the *certification* of safety-critical systems. The design of MC systems is often subject to certification at each criticality level by a certification authority (CA), for example, by Federal Aviation Authority in the US or the European Aviation Safety Agency in Europe for avionics systems. Certification is about verifying that an appropriate level of assurance in meeting the deadlines of the tasks at each criticality level is guaranteed. The level of assurance needed in meeting the deadlines of the tasks may be different at each criticality level. Conventional scheduling strategies, that address both the "criticality" (i.e., multiple WCET of the same task) and "deadline" aspects of MC systems, are not cost- and resource-efficient. Yet another major challenge in the design of MC system is devising a multiprocessors FP scheduling strategy that addresses both the criticality and deadline aspects of the tasks while facilitating certification and efficient resource usage. This challenge is addressed in this thesis along with the challenge of assigning fixed-priorities to the tasks.

Contribution. This thesis proposes a preemptive FP multiprocessor scheduling algorithm, called Mixed-criticality Scheduling algorithm on Multiprocessors (MSM). The MSM algorithm is based on traditional global FP scheduling but with the additional feature of runtime monitoring of the mixed-criticality behavior of the system. The actual

execution time of the tasks at any time instant defines the mixed-criticality behavior of the system at that time instant. When the actual execution time of any task exceeds the WCET estimated for certain criticality level, the system switches to a higher criticality behavior. The system monitors the mixed-criticality behavior at each time instant and dispatches tasks relevant to that criticality behavior based on global FP scheduling strategy. The schedulability analysis of the MSM algorithm derives response-time based schedulability tests to verify the timing constraints at each criticality level. The response-time test of the tasks at each criticality level can be used by the system designers to ensure that the timing constraints for different mixed-criticality behaviors are guaranteed, which facilitates certification.

The proposed response-time based tests are not only applicable to any given fixed-priority ordering of the tasks, they can also be used to find the priority ordering of a given task set. Finding such a priority ordering is required if the test fails for the given priority ordering of the tasks. Simulation results show significant improvement in guaranteeing schedulability of randomly-generated task sets using the proposed searching mechanism for priority assignment over that of using simple[8] priority assignment policy. In contrast to other works on mixed-criticality scheduling, where only two criticality levels are considered, the proposed MSM algorithm and its analysis is applicable for arbitrary criticality levels. This makes the MSM algorithm relevant since many safety-critical systems typically have more than two criticality levels. While a majority of the earlier work consider uniprocessors and dynamic-priority scheduling of MC tasks, the MSM algorithm considers a multiprocessor platform, making it applicable for the emerging CMP technology and the industry-preferred FP scheduling policy.

1.3 Applicability of this Research

The non-functional properties — timeliness, fault-tolerance and mixed-criticality — considered in this thesis are common to many safety-critical real-time systems. While the functional behaviors of different systems are generally different, the modeling and analysis principle of common non-functional behaviors of different systems can be the same. Consequently, the research results presented in this thesis are applicable to a variety of safety-critical real-time systems. For example, the braking function in an automotive system and adjusting the trajectory of a shuttle in the space are completely two different functional behaviors. However, the same scheduling principle might be used for dispatching the control tasks of both functions if the offline analysis and verification of the scheduling algorithm guarantees the timeliness requirement for both functions.

The real-time scheduling algorithms and their analysis presented in this thesis can be used to ensure predictability (in terms of timeliness, fault tolerance and mixed criticality) for safety-critical systems. The approaches proposed in this thesis can help the system designers to efficiently determine offline whether the design constraints needed for

[8] By simple priority assignment it means that the priorities are determined based on heuristics, for example, decreasing periods or decreasing deadlines of the tasks.

acceptable non-functional behaviors of the system are met or not. The proposed schedulability tests can also be used to estimate the resource requirements to satisfy a given set of design constraints. The designers can change the parameters of the mathematical expressions used to represent the schedulability tests to experiment with "what-if" scenarios. This capability enables the designer to make a trade-off between the resource requirement and the rigidity of the design constraints.

All iterative schedulability tests proposed in this thesis assume an arbitrary fixed-priority ordering of the tasks. However, when a task set does not satisfy the schedulability test for a given priority ordering of the tasks, finding another priority ordering (which could make the task set to satisfy the schedulability test) is important since it would not require any changes in hardware, software or specification. The iterative schedulability tests proposed in this thesis can be used to search for such a priority ordering in case the given priority ordering is deemed to be infeasible. This is particularly important for multiprocessors where the optimal priority ordering is currently not known. In summary, the scheduling algorithms and their analysis presented in this thesis have wide applicability for verifying the timing, reliability and mixed-criticality constraints for a variety of safety-critical systems.

Organization of the thesis. The rest of the thesis is organized as follows: Chapter 2 presents the necessary background for real-time computing, fault-tolerance, and mixed-criticality. Chapter 3 presents the system (i.e., task, resource, and fault) model. Chapter 4 outlines the major contributions of the thesis in details. The density-bound-based test and the iterative test for global FP scheduling are presented in Chapter 5 and Chapter 6, respectively. The fault-tolerant scheduling algorithms for uni- and multiprocessors are presented in Chapter 7 and Chapter 8, respectively. Then, Chapter 9 presents the multiprocessor schedulability analysis and response-time test for MC systems. Finally, Chapter 10 concludes the thesis.

2 Preliminaries

In this chapter, the related background and basic concepts of real-time scheduling, fault-tolerance, and mixed-criticality systems are presented.

2.1 Real-Time Systems

Real-time systems are computerized systems with timing constraints. Real-time systems can be classified as *hard real-time systems* and *soft real-time systems*. In hard real-time systems, the consequences of missing a task's deadline may be catastrophic. In soft real-time systems, the consequences of missing a deadline are relatively milder. Examples of hard real-time systems are space applications, fly-by-wire aircraft, radar for tracking missiles, etc. Examples of soft real-time systems are on-line transactions used in airline reservation systems, multimedia systems, etc. This thesis deals with scheduling algorithms and their analysis for hard real-time systems. The most relevant real-time scheduling concepts are: sporadic task system, task priority, preemptive scheduling algorithm, schedulability test, density bound, and so on.

2.1.1 Sporadic Task Systems

The basic component of real-time scheduling is a *task*. The functional behavior of an application is implemented by executing a collection of tasks. The model of a task set captures the workload requirement of an application and the real-time constraints that need to be satisfied for acceptable non-functional behavior. A *sporadic task system* is

a set of tasks in which each task is characterized by three parameters: *minimum inter-arrival time*, *relative deadline* and *worst-case execution time (WCET)*.

Minimum inter-arrival time: Each task in a sporadic task system has a minimum inter-arrival time of occurrence, called the *period*, of the task. The release time of any two consecutive instances, called *jobs*, of a task are separated by at least the period of the task. The *release time* of a job is the instant in time when the job becomes available for execution. A job of a task is *ready* to execute when it is released and remains *active* until it completes its execution. A job of a task is released no earlier than the period plus the release time of the previous job.

Relative Deadline: Each job of a task has a *relative deadline* that is the time by which the job must finish its execution relative to its release time. The relative deadlines of all the jobs of a particular task are the same. The *absolute deadline* of a job is the time instant equal to release time plus the relative deadline.

WCET: Each sporadic task has a worst-case execution time (WCET), which is the maximum CPU time that each job of the task requires in order to complete its execution between its release time and absolute deadline. Determining the exact WCET of a piece of code is challenging and also an active research area [BEL11, GLYY12, LNBCG11, CKR+12, YKS11]. The methodology used to determine the WCET of a piece of code is outside the scope of this thesis. It is assumed that the WCET of each task is known.

If the relative deadline of each task in a task set is less than or equal to its period, then the task set is called a *constrained-deadline* task system. If the relative deadline of each task is exactly equal to its period, then the task set is called an *implicit-deadline* task system. If a sporadic task system is neither constrained nor implicit, then it is called an *arbitrary-deadline* task system. In this thesis, scheduling of constrained-deadline sporadic task system is considered. Since the relative deadline of each task in a constrained-deadline task set is also allowed to be equal to its period, the results presented in this thesis for constrained-deadline task system are also applicable to implicit-deadline sporadic task systems. And, because the jobs of a sporadic task are allowed to be released as quickly as possible, i.e., strictly periodically, the results of this thesis for sporadic task system are also applicable to periodic task systems where successive releases of the jobs are exactly separated by its period.

Task Independence. The execution of the tasks of a real-time application may be dependent on one another, for example, due to resource or precedence constraints. If a resource is shared among multiple tasks, then some tasks may be blocked from being executed until the shared resource is free. Designing better resource sharing protocol for both uni- and multiprocessors is an ongoing research area [BA10, BCB+08, GESY11a, NSBS09]. Similarly, if tasks have precedence constraints, then one task may need to wait until another task finishes its execution. There are many work that consider precedence constraints [SEGY11, SS94, BCGM99]. In this thesis, all tasks are assumed to be independent, that is, there exists no dependency of one tasks on another. The only resource the tasks share is the processor platform.

2.1.2 Task Priority

When two or more ready tasks compete for the use of the processor(s), some rules must be applied to allocate the use of processor(s). This set of rules is often governed by the priority discipline for many real-time scheduling algorithms. The selection of the ready task for execution is determined based on the priorities of the tasks. The priority of a task can be *static* or *dynamic*.

Static Priority: In static (fixed) priority discipline, each task has a priority that never changes during run time. The different jobs of the same task have the same priority relative to any other tasks. For example, according to Liu and Layland, the well known Rate-Monotonic (RM) scheduling algorithm assigns static priorities to tasks such that *the shorter the period of the task, the higher the priority* [LL73]. In preemptive RM scheduling, the task with the shortest period is always dispatched for execution. This thesis considers fixed task priority for all the scheduling algorithms.

Dynamic Priority: In dynamic priority discipline, different jobs of a task may have different priorities relative to the priorities of other tasks in the system. In other words, if the priorities of different jobs of the same task change from one execution to another, then the priority discipline is dynamic[1]. For example, the well known Earliest-Deadline-First (EDF) scheduling algorithm assigns dynamic priorities to tasks such that *a ready job whose absolute deadline is the nearest has the highest priority* [LL73]. In preemptive EDF scheduling, the ready job with the shortest absolute deadline is always dispatched for execution. While EDF is a job-level static priority scheme, the priority assignment scheme governed by pFair scheduling, proposed by Baruah et al. [BCPV96], is a non job-level static priority scheme.

2.1.3 Preemptive Scheduling

A scheduling algorithm is *preemptive* if the release of a new job of a higher priority task can preempt the currently running job of a lower priority task. During runtime, task scheduling is essentially determining the highest priority active job(s) and executing them on the processor(s), possibly by preempting some lower priority job(s).

Under non-preemptive scheme, the job of a currently executing task always completes its execution before another ready job starts execution. A higher priority ready job may need to wait in the ready queue until the currently executing job (may be of lower priority) completes its execution. This will result in worse schedulability performance than for the preemptive case. In this thesis, preemptive scheduling is considered.

2.1.4 Work-Conserving Scheduling

A scheduling algorithm is work conserving if it never idles a processor whenever there is a ready task awaiting execution on that processor. A work conserving scheduler guaran-

[1] In this thesis, static priority means task-level static priority and dynamic priority means job-level static priority. In non job-level static-priority, the same job may have different priorities at different time instants.

tees that whenever a job is ready and the processor for executing the job is free, the job will be dispatched for execution. For example, scheduling algorithms RM and EDF are work-conserving by definition. A non work-conserving algorithm may decide not to execute any task even if there is a ready task awaiting execution. In this thesis, the work-conserving scheduling algorithms are considered.

2.1.5 Schedulability and Optimality

If scheduling algorithm $\mathcal{A}$ can generate a schedule for a given set of tasks such that all the tasks meet their deadlines, then the task set is said to be *schedulable* using that scheduling algorithm $\mathcal{A}$. If a task set is schedulable using scheduling algorithm $\mathcal{A}$, then the task set is *$\mathcal{A}$-schedulable*.

A scheduling algorithm is said to be *optimal*, if it can successfully schedule a task set whenever some other algorithm can schedule the same task set under the same scheduling policy (with respect to, for example, priority assignment discipline, preemptivity, etc.). For example, Liu and Layland [LL73] showed that the RM and EDF are the optimal uniprocessor scheduling algorithms for implicit-deadline tasks under the static and dynamic priority assignment policy, respectively.

While the optimal scheduling algorithm on uniprocessor is known for sporadic task sets, the optimal static or dynamic priority scheduling algorithm for multiprocessors is currently unknown [DB11a]. Optimal multiprocessor scheduling are only known for non-job level static priority discipline (known as pFair family of algorithms [BCPV96, AS04, ZMM03, CRD06]). However, such algorithms suffers from significant number of context-switch and scheduling overheads which make these algorithms impractical to implement without sacrificing some schedulability [HA05].

The notion of optimality is also applicable to a priority-assignment policy under specific scheduling algorithm and processor platform. A fixed-priority assignment policy is said to be optimal if given some fixed-priority assignment policy using which a task set is fixed-priority schedulable on a given platform, then the optimal priority assignment also guarantees the same. For example, the RM and DM are the optimal fixed-priority assignment policies for uniprocessor fixed-priority scheduling of implicit- and constrained-deadline tasks, respectively [LL73].

2.1.6 Schedulability Test

For a given task set, it is computationally impractical to simulate the execution of the tasks at all time instants to see offline whether the task set will be schedulable during runtime. However, the designers of hard real-time systems need to ensure a priori that all the timing constraints are met. To address this problem, schedulability tests for scheduling algorithms are used. A *schedulability test* of a scheduling algorithm $\mathcal{A}$ is a (set of) condition(s) that is (are) used to determine whether a task set is $\mathcal{A}$-schedulable on a particular platform. A schedulability test can be *necessary and sufficient (exact)* or it can be *sufficient* only.

Necessary and Sufficient (Exact) Schedulability Test: A task set will meet all its deadlines if, and only if, it passes the exact test. If the exact schedulability test of a scheduling algorithm $\mathcal{A}$ is satisfied, then the task set is $\mathcal{A}$-schedulable. Conversely, if the task set is $\mathcal{A}$-schedulable, then the exact schedulability condition of algorithm $\mathcal{A}$ is satisfied. Therefore, if the exact schedulability test of a task set is not satisfied, then it is also true that the scheduling algorithm can not successfully schedule the task set.

Deriving an exact test for a scheduling algorithm is always tempting as it guarantees either schedulability or unschedulability of a task set using the corresponding scheduling algorithm. However, deriving an exact test requires precise schedulability analysis considering the worst-case behavior of the algorithm in scheduling a task set. Determining the actual worst case, and then performing precise schedulability analysis, may not be always possible due to lack of time or complexity of the analysis. Therefore, the worst case may need to be safely approximated by introducing some degree of pessimism when analyzing a scheduling algorithm. Introducing such pessimism often results in simpler but sufficient schedulability test.

Sufficient Schedulability Test: A task set will meet all its deadlines if it passes the sufficient test. If the sufficient test of a scheduling algorithm $\mathcal{A}$ is satisfied, then the task set is $\mathcal{A}$-schedulable. However, the converse is not necessarily true. Therefore, if the sufficient schedulability condition of a task set is not satisfied, then the task set may or may not be schedulable using the scheduling algorithm.

Domination. To compare different scheduling algorithms and schedulability tests, the concept of *domination* is useful. Scheduling algorithm $\mathcal{A}$ dominates scheduling algorithm $\mathcal{B}$, if any task set schedulable using algorithm $\mathcal{B}$ is also schedulable using algorithm $\mathcal{A}$, and not conversely. In other words, if scheduling algorithm $\mathcal{A}$ dominates scheduling algorithm $\mathcal{B}$, then all the task sets schedulable using algorithm $\mathcal{B}$ are also schedulable using algorithm $\mathcal{A}$ and there is at least one task set that is not schedulable using algorithm $\mathcal{B}$ but schedulable using algorithm $\mathcal{A}$. Similarly, a schedulability test $\mathcal{P}$ dominates schedulability test $\mathcal{Q}$, if any task set that satisfies test $\mathcal{Q}$ also satisfies test $\mathcal{P}$, and not conversely. In other words, if schedulability test $\mathcal{P}$ dominates schedulability test $\mathcal{Q}$, then all the task sets that satisfy test $\mathcal{Q}$ also satisfy test $\mathcal{P}$ and there is at least one task set that does not satisfy test $\mathcal{Q}$ but satisfies test $\mathcal{P}$.

2.1.7 Minimum Achievable Density

A processor platform is said to be fully utilized when an increase in the density of any of the tasks in an arbitrary constrained-deadline task set will make the task set unschedulable on the platform. The *minimum achievable density* of a scheduling algorithm is the minimum over all total densities of all task sets that fully utilize the processor platform.

A scheduling algorithm can successfully schedule any set of constrained-deadline tasks on a processor platform if the total density of the task set is less than or equal to the minimum achievable density of the scheduling algorithm. The higher the minimum achievable density of a scheduling algorithm, the better is the scheduling algorithm in terms of utilizing the processing resources while meeting the deadlines of the tasks.

Deriving the minimum achievable density may not be always possible due to the pessimism introduced during the schedulability analysis of a scheduling algorithm. However, a bound that is lower than the actual minimum achievable density of a scheduling algorithm can be derived. Such a lower bound on the actual minimum achievable density is simply called a *density bound* of the scheduling algorithm. Since the density bound of a scheduling algorithm is not greater than the minimum achievable density, any task set having total density not greater than the density bound is schedulable using that scheduling algorithm.

Schedulability tests using density bound is called the density-bound-based test. The density-bound-based test compares the total density of a constrained-deadline task set with the density bound to determine whether all the deadlines are met. If the density bound of scheduling algorithm $\mathcal{A}$ is greater than the density bound of scheduling algorithm $\mathcal{B}$, then the density-bound test for scheduling algorithm $\mathcal{A}$ dominates the density-bound test for scheduling algorithm $\mathcal{B}$. In this thesis, a density bound for global FP multiprocessor scheduling is proposed. This proposed test dominates the state-of-the-art density bound for FP scheduling of constrained-deadline sporadic tasks.

If the deadline of each task is equal to its period, then the density bound is called the *utilization bound* and the corresponding schedulability test is given as follow: if the total utilization of a task set is not greater than the utilization bound of a scheduling algorithm, then all the deadlines are met using that scheduling algorithm.

Iterative Schedulability Tests: The density bound test requires exactly one condition to be tested for the entire task set: the total density of a task set is compared with the density bound. On the other hand, an *iterative test* requires one condition to be tested for each task in a task set. The well known response-time test [ABR$^+$93, LSD89, JP86] for uniprocessor fixed-priority scheduling is an example of iterative test where the response time of each task is computed and compared against its relative deadline. The *response time* of a task is the largest time interval between the completion time and release time of any job of the task. Therefore, if the response time of a task is smaller than its relative deadline, then all the jobs of the task meet their deadlines. This thesis proposes new iterative schedulability tests which dominate the state-of-the-art iterative test for constrained-deadline sporadic task sets for global FP scheduling.

2.1.8 Scheduling Algorithms

Scheduling algorithm is a method / policy used to dispatch the jobs of tasks that share some resource, for instance, CPU time on a particular platform. In this thesis, the only resource assumed to be shared among the tasks is the processing platform. Depending on the computing platform, a scheduling algorithms can be categorized as either uniprocessor scheduling or multiprocessor scheduling. In this subsection, the basic principle of preemptive FP scheduling is presented.

Uniprocessor Scheduling. Uniprocessor scheduling algorithm dispatches tasks on a single processor. A uniprocessor FP scheduling algorithm always executes the highest priority active task. If a new job of some task arrives such that its priority is higher than

that of the task currently executing on the processor, then the lower priority (executing) task is preempted and the job of the higher priority task is dispatched for execution. The preempted job may later resume its execution when it becomes the highest priority active job.

Whether the deadlines of a task are met or not depends on the interference caused by the higher priority tasks. The *interference* on a job of a particular task is the cumulative length of intervals during which the job is ready but can not be executed due to the execution of its higher priority tasks. Evidently, the set of higher priority tasks determines the amount of interference on a lower priority task. Consequently, the fixed-priority ordering of the tasks plays an important role in determining the schedulability of each task in a task set. Whether a task set is schedulable under a certain FP assignment can be determined using appropriate schedulability test. Liu and Layland in [LL73] derived a sufficient utilization-bound test for RM scheduling of implicit-deadline tasks on uniprocessors: if the total utilization of a task set is not greater than $n(2^{\frac{1}{n}} - 1)$, then all the tasks meets their deadlines, where n is the number of tasks in a task set. Necessary and sufficient (exact) schedulability test for uniprocessor FP scheduling have been derived in [LSD89, JP86, ABR$^+$93, ABRW91]. The exact test proposed in [ABR$^+$93] for uniprocessor DM scheduling is presented in Subsection 7.2.1 (page 117).

Multiprocessor Scheduling. In multiprocessor scheduling, tasks can be scheduled using one of the two basic multiprocessor scheduling principles: the *global* scheduling and the *partitioned* scheduling. In global scheduling, a task is allowed to execute on any processor (even when it is resumed after preemption). This is done by keeping all the ready tasks in a global queue from which the highest priority tasks are dispatched to the processors, possibly by preempting some lower priority tasks, based on fixed priority assigned to each task.

In partitioned scheduling, the task set is grouped in different task partitions and each partition has a fixed processor onto which all the tasks of that partition are assigned. A *task assignment algorithm* partitions the task set and decides the mapping of each task to a particular processor. In partitioned scheduling, ready tasks assigned in one processor are not allowed to execute in another processor even if the other processor is idle. Evidently, tasks can migrate in global scheduling while no migration is allowed in partitioned scheduling. The advantage of partitioned scheduling is that once tasks are assigned to processors, each processor can execute tasks based on mature uniprocessor scheduling algorithms. Many static-priority scheduling policies for both global [ABJ01, Lun02, Bak06, BG03b, BCL05, And08a, DB11b, DB09] and partitioned [DL78, AJ03, FBB06, LMM98a, LBOS95, LDG04, LGDG03, OB98, OS95b] approaches have been studied to derive appropriate schedulability tests.

The main goal of schedulability analysis for many global and partitioned FP scheduling algorithms is to derive a *schedulability test* that — when satisfied for a given task set — implies that all the deadlines are met. It has already been proved that there exists some implicit-deadline task set with utilization slightly greater than 50% of the capacity of a multiprocessor platform on which a deadline miss must occur for both global and partitioned static-priority scheduling [ABJ01, OB98]. Therefore, the minimum achievable

utilization for both global and partitioned multiprocessor scheduling can not be greater than 50%. Moreover, it is also well-known that applying the uniprocessor RM scheme to multiprocessor global scheduling can lead to missed deadlines of tasks even when the utilization of a task set is close to 0% of the capacity of the multiprocessor platform. This effect is known as *Dhall's effect* [DL78, Dha77]: some task with large utilization is assigned lower RM priority and misses its deadline.

Technique to avoid Dhall's effect for static-priority is first proposed in [ABJ01] which is further improved in [Lun02, BCL05, And08a]. Luckily, *Dhall's effect* is absent in partitioned scheduling. The main challenge for partitioned scheduling is instead to develop an efficient task assignment algorithm for partitioning a task set. However, since the problem of determining whether a schedulable partition exists is an NP-hard problem [GJ79], different heuristics have been proposed for assigning tasks to multiprocessors using partitioned scheduling. The majority of the heuristics for partitioned scheduling are based on different bin-packing algorithms (such as First-Fit or Next-Fit [LDG04]). One such bin-packing heuristic is First-Fit (FF), which is described next.

First-Fit (FF) Heuristic. With the FF heuristic, all the processors (e.g. processor one, processor two, and so on) and tasks (task one, task two and so on) are indexed. Tasks may be indexed based on some ordering of the task parameters (for example, sort the task set based on increasing/decreasing periods or utilizations) or can simply follow any arbitrary ordering for indexing. For example, Dhall and Liu in [DL78] proposed FF partitioned scheduling using the sufficient RM schedulability test where tasks are first sorted based on increasing periods. Starting with the task with lowest index, tasks are assigned to the lowest-indexed processor, always starting with the first processor (processor one). To determine if an unassigned task will be schedulable on a particular processor, when assigned along with the already-assigned tasks on that processor, a uniprocessor schedulability test is used. If a task can not be assigned to the first processor based on that schedulability test, then the task is considered to assign in the second processor, and so on. If all the tasks are assigned, then the partitioning of the task set is successful. If some task can not be assigned to any processor, then the task set can not be partitioned using FF.

Task-Splitting Algorithms. The different degrees of migration freedom for tasks in the global and partitioned scheduling can be considered as two extremes of multiprocessor scheduling. While in global scheduling no restriction is placed for task migration from one processor to another, partitioned scheduling disallows migration completely. This strict non-migratory characteristic of partitioned multiprocessor scheduling is relaxed using a promising concept called *task-splitting* in which some tasks, called *split-tasks*, are allowed to migrate to a different processor. Task splitting does not mean dividing the code of the tasks; rather it is migration of execution of the split tasks from one processor to another. Recent research has shown that task splitting can provide better performance in terms of schedulability and can overcome the limitations of minimum achievable utilization of 50% for pure partitioned scheduling [AT06, AB08, ABB08, KY08, KY09, GSYY10, LRL09, BBA11, PJ10].

2.2 Fault-Tolerant Systems

A fault-tolerant system is one that continues to perform its specified service in the presence of hardware and/or software faults. In designing fault-tolerant systems, mechanisms must be provided to ensure the correctness of the expected service even in the presence of faults. Due to the real-time nature of many fault-tolerant systems, it is essential that the fault-tolerance mechanisms provided in such systems do not compromise the timing constraints of the real-time applications. In this section, the basic concepts of fault-tolerant systems under the umbrella of real-time systems are discussed.

2.2.1 Failure, Error, and Fault

Avižienis and others define the terms *failure*, *error* and *faults* in [ALRL04].

Failure A system *failure* occurs when the service provided by the system deviates from the specified service. For example, when a user can not read his stored file from computer memory, then the expected service is not provided by the system.

Error An *error* is a perturbation of internal state of the system that may lead to failure. A failure occurs when the erroneous state causes an incorrect service to be delivered, for example, when certain portion of the computer memory is corrupted or broken and the stored files therefore can not be read by the user.

Fault The cause of the error is called a *fault*. An active fault leads to an error; otherwise the fault is dormant. For example, impurities in the semiconductor devices may cause computer memory in the long run to behave unpredictably.

If a fault remains dormant during system operation, then there is no error. If the fault leads to an error, then it must be tolerated so that the error does not lead to system failure[2]. Identifying the characteristics of the faults and the corresponding errors is an important issue for the design of an effective fault-tolerant system. Faults in systems may be introduced during development (for example, design and production faults) or due to the interaction with the external environment (for example, faults entering via user interface or due to natural process such as radiation). Based on persistence, faults can further be classified as permanent, intermittent, and transient [Joh88]. Faults can occur in hardware or/and software.

Hardware Faults: A permanent failure of the hardware is an erroneous state that is continuous and stable. Such erroneous state is caused by some permanent fault in the hardware. On the other hand, transient hardware faults are temporary malfunctioning of the computing unit or any other associated components which causes incorrect state in the system. Intermittent faults are repeated occurrences of transient faults. Transient faults and intermittent faults manifest themselves in a similar manner. They happen for a short time and then disappear without causing a permanent damage. If the error caused

[2]Example 1.1 (page 10) demonstrates the terms — faults, errors and failures — using an example.

by such transient faults are recovered, then it is expected that the same error will not reappear since transient faults are short lived. To tolerate a permanent processor failure, either the processor is repaired / replaced or its effect is mitigated by executing the task on a redundant processor.

- **Sources of Hardware Transient Faults:** The main sources of transient faults in hardware are environmental disturbances like power fluctuations, electromagnetic interference and ionization particles. Transient faults are the most common, and their number is continuously increasing due to high complexity, smaller transistor sizes and low operating voltage for computer electronics [Bau05].

- **Rate of Transient Faults:** It has been shown that transient faults are significantly more frequent than permanent faults [SKM+78, CMS82, IRH86, CMR92, Bau05, SABR04]. Siewiorek and others in [SKM+78] observed that transient faults are 30 times more frequent than permanent faults. Similar result is also observed by Castillo, McConnel and Siewiorek in [CMS82]. In an experiment, Iyer and others found that 83% of all faults were determined to be transient or intermittent [IRH86]. The results of these studies show the need to design fault-tolerant system to tolerate transient faults.

  Experiments by Campbell, McDonald, and Ray using an orbiting satellite containing a microelectronics test system found that, within a small time interval ($\sim$ 15 minutes), the number of errors due to transient faults is quite high [CMR92]. The result of this study shows that in space applications, the rate of transient faults could be quite high and a mechanism is needed to tolerate multiple transient faults within a particular time interval. It was shown in [SKK+02] that the error rate in processors due to transient faults is likely to increase by as much as eight orders of magnitude in the next decade. Moreover, given the fact that transistor size and operating voltage are shrinking for recent computer electronics, the number of transient faults is expected to rise in future within a given time interval.

Software Faults: All software faults, known as software bugs, are permanent. However, the way software faults are manifested as errors leads to categorize the effect as: permanent and transient errors. If the effect of a software fault is *always* manifested, then the error is categorized as permanent. For example, initializing some global variable with incorrect value which is always used to compute the output is an example of a permanent software error. On the other hand, if the effect of a software fault is not always manifested, then the error is categorized as transient. Such transient error may be manifested in one particular execution of the software and may not manifest at all in another execution. For example, when the execution path of a software varies based on the input (for example, sensor values) or the environment, a fault that is present in one particular execution path may manifest itself as an transient error only when certain input values are used. This fault may remain dormant when a different execution path is taken, for example, due to a change in the input values or environment.

The fault-tolerant scheduling algorithms proposed in this thesis considers tolerating multiple task errors within a time interval equal to the largest relative deadline of the tasks in a sporadic task set. Such task errors may be caused by software faults or transient hardware faults. In addition, the fault-tolerant scheduling algorithm proposed for multiprocessors also considers tolerating[3] permanent processor failures. The fault model considered for processor failures is permanent hardware faults that are continuous and stable. Processors are assumed to be *fail-stop* processors: each processor is either working correctly or ceases functioning [SS83, Sch84].

2.2.2 Error Detection Techniques

In order to tolerate a fault that leads to an error, fault-tolerant systems rely on effective error detection mechanisms. The design of many fault-tolerant scheduling algorithm relies on effective mechanisms to detect errors. Error detection mechanisms and their coverage (e.g., percentage of errors that are detected) determine the effectiveness of the fault-tolerant scheduling algorithms.

Error detection can be implemented in hardware or software. Hardware implemented error detection can be achieved by executing the same task on two processors and compare their outputs for discrepancies (*duplication and comparison technique using hardware redundancy*). Another cost-efficient approach based on hardware is to use a watchdog processor that monitors the control flow or performs reasonableness checks on the output of the main processor [MCS91]. Control flow checks are done by verifying the stored signature of the program control flow with the actual program control flow during runtime. In addition, today's modern microprocessors have many built-in error detection capabilities like, error detection in memory, cache, registers, illegal op-code detection, and so on [MBS07, WEMR04, SKK+08, KSSF10].

There are many software-implemented error-detection mechanisms: for example, executable assertions, time or information redundancy-based checks, timing and control flow checks, and etc. Executable assertions are small code in the program that checks the reasonableness of the output or value of the variables during program execution based on the system specification [JHCS02]. In time redundancy, an instruction, a function or a task is executed twice and the results are compared to allow errors to be detected (*duplication and comparison technique used in software*) [AFK05]. Additional data (for example, error-detecting codes or duplicated variables) are used to detect occurrences of an error using information redundancy [Pra07].

In summary, there are numerous ways to detect the errors and a complete discussion is beyond the scope of this thesis. The fault-tolerant scheduling algorithms proposed in this thesis rely on effective error-detection mechanisms.

[3] By "tolerating" it does not mean preventing/stopping the failure in some way; rather, it means that the effect of permanent processor failure is mitigated by executing the tasks on other non-faulty processors.

2.3 Mixed-Criticality Systems

An MC system is defined as follows in [BBB+]:

> "A mixed-critical system is an integrated suite of hardware, operating system and middleware services and application software that supports the execution of safety-critical, mission-critical, and non-critical software within a single, secure compute platform."

In short, an MC system is one in which the functionalities hosted on a common platform have different criticality levels. For example, in the RTCA DO-178B standard, there are five different Design Assurance Levels (DAL A to DAL E) for software in avionics systems (please see Table 2.1). The "criticality" of a function or task specifies its "importance". The consequence for not meeting the specification of a high critical function could be severe. The criticality assigned to a function specifies the level of assurance or confidence needed regarding the correct behavior of the function.

Level	Failure Condition	Interpretation
A	Catastrophic	Software that could cause or contribute to the failure of the system resulting in the loss of ability to continue safe flight and landing. Failures may cause a crash. An example of such system is an engine controller software.
B	Hazardous	Software that could cause or contribute to the failure of the system resulting in serious or fatal injuries to the aircraft occupants. Examples is pressurization system software.
C	Major	Software that could result in a major failure condition or discomfort to the occupants of the aircraft.
D	Minor	Failures results in some inconvenience to the occupants of the aircraft. Example is failure causing a routine flight plan change.
E	No Effect	Software that could cause or contribute to the failure of the system resulting in no effect on the system. Examples are entertainment system, Internet access.

Table 2.1: *RTCA published the DO-178B software development process standard "Software Considerations in Airborne Systems and Equipment Certification". The United States Federal Aviation Authority (FAA) accepts the use of DO-178B as a means of certifying software in avionics application. The five DO-178B levels describe the consequences of a potential failure of the software: catastrophic, hazardous, major, minor, or no-effect.*

The need for research in the domain of MC systems is motivated in [BBB+] using an example of Unmanned Aerial Vehicle (UAV) which is expected to operate over or close to civilian airspace. Such a system has both *flight-critical* and *mission-critical* functionalities that require safety, reliability and timeliness guarantee. In addition, such a system must pass the mandatory certification from standard civil aviation authority. Certification of MC system is challenging and costly approach since such system is relatively complex due to the integration of functionalities with different criticality levels.

The objective of designing an MC systems is to combine previously independent system applications into a single computation platform while also ensuring that the system is predictable. In other words, the integration of mixed-critical functions on a common computing platform aims to save cost while at the same time hope to improve the overall performance in terms of, for example, safety, reliability and timeliness. Research in the real-time community has recently received considerable attention considering two important factors of MC systems: (i) run-time robustness, and (ii) design-for-certification.

Run-time Robustness. One of the most important requirement for designing mixed-criticality systems is in ensuring the non-interference or isolation property among functions of different criticality levels. In particular, a high-critical function must not be adversely affected by a low-critical function. In the context of real-time scheduling, temporal isolation is achieved by ensuring that if the system is not capable of meeting some deadline (e.g., due to overload situation), then no deadline of a high-criticality task is missed before all the low-critical tasks.

Physical separation of resources is one option to achieve the isolation property where the functionalities including all logic and processor are physically separated. For example, the safety-critical functions, e.g., flight control, engine control, electrical power system control in an UAV may have their own hardware, software, and standard interfaces to communicate with other functions.

Due to the space, weight, and power considerations, providing such dedicated resources is costly or may even impractical for many resource-constrained systems. To solve this problem, integration of multiple functionalities on the same platform is considered where the isolation property is achieved by partitioning the system resources. For example, according to ARINC 653 standard, the system must provide space and temporal partitioning of all resources — e.g., memory, processing time, communication bus — for all the hosted functionalities. In such an approach, a partition allocated to a low-critical function can not be used by a high-critical function. Therefore, a high-critical task may miss the deadline in its time partition while a low-critical task meet the deadline in another time partition (known as criticality inversion [NLR09]). Moreover, resources are not utilized efficiently since function in one partition is not allowed to execute on a different partition. To avoid the criticality inversion problem and to efficiently use the processing resources, "true" sharing of the platform is considered while providing run-time robustness. New resource allocation and scheduling algorithms are being designed such that the system provides run-time robustness [NLR09, LdNRM10, TSP11, YYP+12].

Design-for-Certification: Another important aspect of MC system, which is addressed in this thesis, is design-for-certification. For example, the design and development of an UAV needs to be certified by standard statutory certification authority (CA), for example, by Federal Aviation Authority in the US or the European Aviation Safety Agency in Europe for avionics systems. The CA certifies a system as correct if the assumptions of the CA regarding the system behavior hold at run-time.

Traditionally, when functions having different criticality levels are hosted on the same computing platform, then the system is certified by assuming the highest criticality level for all the functions. Such assumption is pessimistic because certifying at the highest criticality level implies the highest degree of assurance regarding the correct behavior of all the functions which in turn could be guaranteed by over-provisioning the required resources. Therefore, it is necessary to develop new design and analysis techniques that addresses certification of MC systems while efficiently utilizing the processing resources. This thesis proposes fixed-priority scheduling algorithms considering this design-for-certification issue of MC systems.

3 Models

The design and analysis of hard real-time scheduling algorithms is based on appropriate modeling of the target system. This is because a priori knowledge of the workload and available resources is necessary to analyze and ensure predictability of the system. The *task*, *resource* and *fault* models are presented in this chapter. A task model specifies the workload and timing constraints of the real-time application. A resource model specifies the type and capacity of the available resources (e.g., processors) for executing the tasks. A fault model specifies the nature and frequency of faults that the system needs to tolerate.

3.1 Task Model

The formal notations and important concepts of sporadic tasks are now presented.

Sporadic task set. In this thesis, real-time scheduling of n constrained-deadline sporadic tasks in set $\Gamma = \{\tau_1 \dots \tau_n\}$ is considered. Each of the tasks $\tau_i \in \Gamma$ is characterized by a triple (C_i, D_i, T_i), where

- C_i represents the worst-case execution time (WCET) of each job of the task;
- D_i is the relative deadline;
- T_i is the period which is minimum inter-arrival time of the jobs of the task.

Jobs of Tasks. Successive arrivals of the instances or jobs of task τ_i are separated by at least T_i time units. The j^{th} job of task τ_i is denoted by J_i^j. The release time of job

J_i^j is denoted by r_i^j. A job of a task τ_i is released no earlier than the release time of the previous job plus the period T_i, i.e., $r_i^{j+1} \geq (r_i^j + T_i)$. The absolute deadline of job J_i^j is denoted by d_i^j and given as follows:

$$d_i^j = r_i^j + D_i \tag{3.1}$$

A job J_i^j requires at most C_i units of execution time between its release time r_i^j and deadline d_i^j. If task τ_i is periodic and first released at time 0, then $r_i^j = (j-1) \cdot T_i$.

Density and Utilization. The *density* δ_i and *utilization* u_i of a task τ_i are denoted by

$$\delta_i = C_i/D_i$$

$$u_i = C_i/T_i$$

The *total density (resp. utilization)* of task set $\mathcal{A}$ is $\sum_{\tau_i \in \mathcal{A}} \delta_i$ (resp. $\sum_{\tau_i \in \mathcal{A}} u_i$).

Fixed-Priority. For a given fixed-priority ordering of the tasks, the set of tasks with a priority higher than the priority of task τ_i is denoted by HP_i. There are many policies for assigning the fixed-priorities to the tasks. Some example fixed-priority assignment policies are the following:

- **Rate-Monotonic (RM) priority:** The priority of task τ_i is greater than the priority of task τ_j if $T_i < T_j$. This is the priority assignment governed by the RM scheduling policy: a task with smaller period has higher priority.

- **Deadline-Monotonic (DM) priority:** The priority of task τ_i is greater than the priority of task τ_j if $D_i < D_j$. This is the priority assignment governed by the DM scheduling policy: a task with smaller relative deadline has higher priority.

- **Slack-Monotonic (SM) priority:** The priority of task τ_i is greater than the priority of task τ_j if $(D_i - C_i) < (D_j - C_j)$. This is the priority assignment governed by the SM scheduling policy: a task with smaller *slack* has higher priority.

- **Audsley's Optimal Priority Assignment Algorithm:** While the optimal fixed-priority ordering for some system model[1] can be given using simple heuristic (e.g., the DM priority ordering is the optimal for constrained-deadline tasks on uniprocessor [LW82]), the optimal priority ordering for other system model is not necessarily based on simple heuristic. For example, the optimal priority ordering of constrained-deadline tasks having arbitrary start times (offsets) is not necessarily the DM priority assignment policy for uniprocessor. To assign the fixed priorities to such task sets with offsets, an optimal priority assignment (OPA) algorithm, known as Audsley's OPA algorithm, is proposed in [Aud01, Aud91]. Although the OPA algorithm is first proposed for uniprocessors, it has been adapted by Davis and Burns [DB09] for priority assignment on multiprocessors. The basic

[1]The system model consists of the task, resource, and scheduler models.

idea of Audsley's OPA algorithm to assign the priorities is based on a schedulability test S and involves the following steps:

- Initially, no task is assigned any priority (each task is called a priority-unassigned task);
- The fixed priorities are assigned starting from the lowest priority level to the highest priority level, i.e., the task which is first assigned a priority is the lowest priority task and the task which is assigned the final priority is the highest priority task;
- For a particular priority level (staring from the lowest), if any one of the priority-unassigned tasks, sat task τ, is deemed schedulable using the schedulability test S at that priority level, by assuming all other priority-unassigned tasks having higher priorities, then task τ is assigned that priority level;
- If no priority-unassigned task can be assigned the priority level then priority assignment fails. If each task is assigned one priority level, then the priority assignment succeeds.

While many work consider the priority assignment problem and schedulability testing problem as two independent problems, the Audsley's OPA algorithm combines the problem of finding the priority assignment with the schedulability test of each task. Consequently, if all the tasks are assigned priorities according to the OPA algorithm using schedulability test S, then the task set is also schedulable.

In order to find a priority ordering for which a task set passes the schedulability test S, a naive and exhaustive approach is to consider all the $n!$ different priority orderings of the tasks. In contrast, Audsley's OPA algorithm needs to check at most $(n^2 + n)/2$ different priority orderings. Details on OPA algorithm and its applicability to multiprocessor scheduling are presented in Chapter 6.

Time Division. Even though length of time intervals, time instants are often modeled using real numbers, time is not infinitely divisible in actual implementation of a system. The difference in time between occurrence of different events can not be determined more precisely than one tick of the system clock. In this thesis, all time values (e.g, WCET, deadline, and interval length) are assumed to be positive integers.

Critical Instant. The critical instant of a task is the release time at which the interference on the task from the higher priority tasks is maximized. Consequently, the response time of the job released at critical instant is the worst-case response time of the task. Therefore, a job released at a critical instant is schedulable if and only if the task set is schedulable. Liu and Layland have proved that the critical instant for uniprocessor FP scheduling of any task occurs when the task is released simultaneously with the release of all its higher priority tasks [LL73].

The analysis of the proposed fault-tolerant scheduling algorithm `FTDM` in this thesis for uniprocessor platform also considers the critical instant of each task to derive an

exact schedulability test. Under fault-tolerant FP scheduling on uniprocessors, where time redundancy is used to recover from task error, there is one job of each task during execution of which the occurrence of faults have the greatest impact. In such case, the errors may occur in that particular job of the task and/or in any job of its higher priority tasks. To recover from the errors in such situation, the execution of the backups causes the response-time of that particular job to be the maximum.

Ghosh *et al.* showed that, when faults occur and time redundancy is used to tolerate faults in uniprocessor RM scheduling, the critical instant is when all tasks are released simultaneously [GMMS98]. This result can be easily extended for FTDM scheduling (i.e., DM fault-tolerant scheduling on uniprocessor) as follows: if the completion of job J of a task is delayed by Δ time units due to the occurrence of some faults in J or in its higher-priority jobs, then some other lower priority job J' of some other task will be delayed by at most Δ time unit if both J and J' are released simultaneously. Therefore, the exact schedulability test for the proposed FTDM scheduling considers that all the tasks are released at the same time and without loss of generality it is assumed that all the tasks are released at time zero.

Unfortunately, the critical instant which is known for uniprocessor FP scheduling is not applicable to global FP scheduling. Lauzac et. al showed that a task does not have its worst-case response time when released simultaneously with all the higher priority tasks under the global FP scheduling [LMM98b]. In multiprocessor scheduling, the response time of a job that is released simultaneously with all other higher priority tasks may not be the largest because this scenario may not cause all the processors to be busy executing the higher priority tasks for the largest time interval (i.e., interference) over which a lower priority task is awaiting execution. This is demonstrated using the following example.

Example 3.1. *Consider four sporadic tasks with parameters* (C_i, D_i, T_i) *as follows:* $\tau_1(1,1,4)$*,* $\tau_2(1,2,5)$*,* $\tau_3(2,3,4)$*, and* $\tau_4(1,4,4)$*. Assume that tasks are given deadline-monotonic priorities and scheduled on* $m = 2$ *processors using global* FP *scheduling. Also assume that all tasks are simultaneously released at time zero and all jobs arrive as quickly as possible (i.e., strictly periodically). The global* FP *scheduling of these tasks is shown in Figure 3.1.*

The first job J_4^1 *of the lowest priority task* τ_4 *completes its execution at time* $t = 2$ *(response-time is 2 time units). However, the second job* J_4^2 *of task* τ_4 *is released at time* $t = 4$ *and completes its execution at* $t = 7$ *(response-time is 3 time units). Consequently, the worst-case response time of task* τ_4 *is not necessarily equal to the response time of its first job. In other words, the critical instant of a task is not the time instant when all the higher priority tasks are released at the same time in global* FP *scheduling.* □

Given the sporadic nature of the tasks, finding the job that suffers the maximum interference due to the execution of the higher priority jobs can not be determined easily for global FP scheduling. Not knowing the critical instant for analyzing global FP schedulability of a task requires some pessimism to be introduced in the schedulability analysis. As will be evident later, introduction of such pessimism during schedulability analysis results in sufficient schedulability test for global FP scheduling.

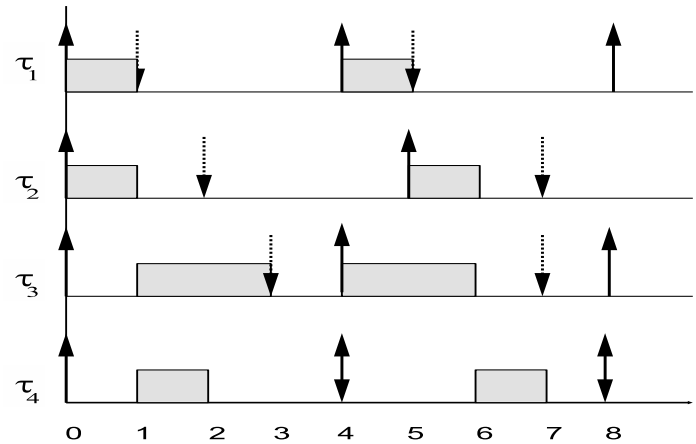

Figure 3.1: *The release time and deadline of each job is shown using upward and downward arrow, respectively. The second job of task τ_4 has larger response time than its first job. Critical instant for global* FP *scheduling is not the instant when all the tasks are released at the same time.*

3.2 Resource Model

In this thesis, the only resource the tasks are assumed to share is the computing platform. Scheduling on multiprocessors considers the availability of m identical unit-capacity processors. In this thesis, multiprocessors and multicores are synonymous since the proposed schedulability analysis and theory for multiprocessors is also applicable to multicores having m identical cores hosted on the same chip.

Task preemptions, migrations, context-switches, scheduling decisions incurs overhead and are extrinsic to the task model at hand. The costs of such different kinds of overhead are assumed to be included in the WCET of each task. This is because, at least for now, there is no analytical method available to calculate the cost of such overheads for sporadic task systems considering different processor architectures, operating systems, and so on. There have been effort to calculate such overheads for specific architecture and operating system based on *empirical* study using strictly periodic task systems [BCA08, BBA10]. Moreover, the preemption and migration overhead due to the loss of cache affinity is dependent on the working set size of individual task. And, the working set size of different tasks of different applications are different. Although such issues are not addressed in this thesis, one can rely on experimental studies (similar to [BCA08, BBA10]) to measure these overhead costs considering the application, operating system, and the target hardware platform. The designer of a real-time system can inflate the WCET of each task after experimentally measuring the cost of different overheads by executing the tasks on the target platform.

3.3 Fault Model

Designing fault-tolerant scheduling algorithm needs to guarantee that all the tasks deadlines are met when faults occur even under the worst-case load condition. No fault-tolerant system, however, can tolerate an infinite number and arbitrary types of faults within a particular time interval based on time redundancy. The scheduling guarantee in fault-tolerant system is thus given under the assumption of a certain fault model.

This thesis considers (i) tolerating task errors for uniprocessor scheduling, and (ii) tolerating both task errors and processor failures for multiprocessor scheduling. The proposed uniprocessor fault-tolerant scheduling algorithm FTDM considers tolerating f task errors within all possible time intervals of length D_{max} where D_{max} is the maximum relative deadline of any task in the task set. The proposed multiprocessor fault-tolerant scheduling algorithm FTGS considers tolerating f task errors within all possible time intervals of length D_{max} and also considers tolerating at most ρ permanent processor failures during the lifetime of the system.

In this thesis, the fault model considered is very strong in the sense that multiple faults (that cause errors) can occur at any time, in any task, and even during the execution of the backups. The faults can also occur in bursts; however, the number of task errors that can be recovered is bounded by f within any possible interval of length D_{max}.

The fault model considers tolerating transient hardware faults due to which the task error is also transient. Transient errors are short lived and would not reappear upon re-executing the task. This is a reasonable assumption since it can be implemented simply by resetting the processor before re-execution. The fault model also considers software fault due to which the task error is transient. When a software fault is manifested as a transient error, then such error can be recovered using simple re-execution. In such case, it is expected that the same error would not occur if the software (task) is simply re-executed. Software faults that result in permanent task errors are also considered in the fault model. If the effect of a software fault is manifested as a permanent error, then simple re-execution of the same task can not mitigate such permanent erroneous behavior. In such case, a diverse implementation of the task has to be executed as backup to recover from the error and such backup may have different WCET than the primary.

Tolerating permanent processor failure is also considered in the fault model for the proposed fault-tolerant scheduling on multiprocessors. The effect of such failures is mitigated by executing the backup of the task that was executing on the faulty processor on a different (non-faulty) processor. The fault model for permanent processor failures covers those hardware faults that are continuous/stable and causes permanent error. Each of the processors in a multiprocessor system is assumed to be fail-stop processors: it is either working correctly or ceases functioning.

If a system is designed to tolerate transient error or permanent processor failures, then either re-execution or diverse backup is effective. However, if the system also needs to tolerate permanent error due to software faults, then all the backups must be different (i.e., implemented diversely) and we have to pay for this costly approach for tolerating such software faults using time redundancy.

Time redundancy is considered for tolerating multiple faults. When fault occurs during execution of a task and an error is detected, either the faulty task is simply re-executed or a diverse implementation of the same task is executed. The diverse implementation of the same task is considered to achieve diversity as is used in N-version programming [Avi85]. A backup of a task, which is a diverse implementation, has the same period, priority, and deadline as the original task but may have a different WCET than the primary. The schedulability analysis of the fault-tolerant algorithms has to consider such different WCETs for different backups of the same task.

In order to tolerate task errors even during the recovery operations (i.e., when a backup is executing) multiple backups are considered for each task. The multiple backups of the same task are ordered based on some design decision (i.e., the first backup is executed whenever the primary fails, the second backup is executed whenever the first backup fails, and so on). For example, the system designer may prefer to run a particular implementation of a task as the primary, and then another implementation (e.g., an exception handler) as the backup if an error is detected in the primary, and so on.

An error is assumed to be detected at the end of execution of a task's primary or backup. This is required for the worst-case schedulability analysis since the detection of an error at the end of execution corresponds to larger wasted CPU time in comparison to the situation when the error is detected in the middle of the execution. There is no fault propagation: one fault is assumed to affect at most one job, either its primary or one of its backups. It is also assumed that, during the execution of each primary or backup of a task, at most one fault could affect this execution. This assumption is also essential for the worst-case schedulability analysis because the overhead for executing the backup, after an error is detected, does not depend on the number of errors affecting that particular primary or backup. If more than one error affect a task's primary or backup, then only one additional backup is activated to recover from those task errors.

Both the proposed `FTDM` and `FTGS` scheduling algorithms consider tolerating f task errors in each of the all possible intervals of length D_{max}. Within any time interval of length D_{max}, the f task errors may occur in the same job of a task — affecting that job's primary and backups. Therefore, the task model is extended to consider f different backups for each task. The WCET of the primary copy of task τ_i is C_i and the WCET of each of the f backup copies of task τ_i is denoted by E_i^k for $k = 1, 2, \ldots f$. All the jobs of the same task have the same WCET for the primary copies and the WCET of the k^{th} backup copy of different jobs of the same task are equal for $k = 1, 2, \ldots f$. If a task errors are detected in job J_i^j (one error in the primary copy and $(a-1)$ errors in the backup copies), the total execution requirement for job J_i^j is $C_i + \sum_{k=1}^{a} E_i^k$. Note that for a maximum of a task errors affecting a particular job J_i^j, the a^{th} backup copy is the non-faulty execution of job J_i^j under the assumed fault model. Moreover, the following must hold for each constrained-deadline task $\tau_i \in \Gamma$ for all $i = 1, 2, \ldots n$:

$$C_i + \sum_{k=1}^{f} E_i^k \leq D_i \tag{3.2}$$

It is assumed that a combination of software and hardware error-detection mechanisms are available to detect task error. There are many software and hardware based error-detection mechanisms as is discussed in Section 2.2.2. Perfect error detection coverage is assumed for simplicity of the schedulability analysis. However, a probabilistic analysis of fault-tolerant schedulability with imperfect error detection coverage can be addressed similar to [BPSW99] and such an analysis is not the addressed in this thesis. It is also assumed that the error-detection and fault-tolerance mechanisms are themselves fault-tolerant. The error detection overhead is considered as part of the WCET of the task. In summary, the fault model considered in this thesis has reasonable representativity and very general to tolerate a variety of faults in hardware/software.

4
Goals and Contributions

The complexity of hardware and software in computerized system is increasing due to the "push-pull" effect between the development of new software for existing hardware and the advancement in hardware technology for forthcoming software. On one hand, high-speed processors pull the development of new software with more functionalities (possibly with added complexities), and on the other hand, application software with new functionalities push the industry to come up with more powerful processors (with added complexities).

Due to the increased complexity of real-time systems both in terms of hardware and software, the design of such systems is becoming more challenging. One of the main challenges is to utilize the processing platform efficiently while satisfying all the timing constraints of real-time systems. The increasing frequency of the occurrences of transient faults in increasingly-complex hardware and the increasing likelihood of having more bugs in complex software require effective and cost-efficient fault-tolerant mechanisms in today's computerized systems. Due to the size, weight and power constraints in many safety-critical embedded systems, the integration of multiple functionalities having different criticality levels on the same hardware platform requires developing new criticality- and certification-cognizant scheduling algorithms. In order to ensure that the non-functional behaviors of real-time systems are acceptable, the design of the system requires appropriate modeling, effective analysis, and proper verification.

The overall goal of this thesis, considering the research questions **Q1**, **Q2** and **Q3** in Section 1.1 (page 3), is to design and analyze resource-efficient scheduling algorithms that can be used to satisfy the timing, reliability and criticality constraints. The major contributions to achieve this goal in this thesis are listed below (contributions **C1** and **C2** address **Q1**; contributions **C3** and **C4** address **Q2**, and contribution **C5** addresses **Q3**):

C1 Density-Bound-Based Test (Chapter 5) – A new fixed-priority assignment policy, called ISM-DS, for constrained-deadline sporadic tasks is proposed. The proposed priority assignment policy addresses the problem of determining the fixed-priority ordering of the sporadic tasks to be scheduled using global FP scheduling. According to the ISM-DS policy, a subset of the tasks (referred to as heavy tasks) is assigned the highest fixed priority and the remaining tasks (referred to as light tasks) are assigned slack-monotonic priorities. The density threshold, based on which a task is classified as being either heavy or light, is calculated based on the number of processors. In order to address the schedulability testing problem, a sufficient density-bound-based schedulability test is derived by analyzing global FP scheduling using the ISM-DS priority assignment policy. This test is shown to dominate the density-bound-based state-of-the-art schedulability test for global FP scheduling of constrained-deadline tasks.

Based on schedulability analysis of the ISM-DS policy, another priority assignment policy, called ISM-DS$[\xi]$, is proposed. Policy ISM-DS$[\xi]$ assigns fixed-priorities to the tasks in a way that is similar to the ISM-DS policy, but using a threshold density ξ which is selected from the set of all the densities of the tasks. To address the schedulability testing problem, the threshold density ξ is selected in such a way that the task set become schedulable using global FP scheduling. It is also proved that the schedulability test for global FP scheduling based on the ISM-DS$[\xi]$ priority assignment policy dominates the density-bound test for the ISM-DS priority assignment policy. Simulation results show that the fraction of randomly-generated task sets deemed schedulable using the schedulability test for the ISM-DS$[\xi]$ priority-assignment policy is significantly higher than that of the density-bound test for the ISM-DS priority assignment policy.

C2 Iterative Test (Chapter 6) – A new response-time-based iterative test, called the IA-RT test, is proposed for global FP scheduling of constrained-deadline sporadic tasks. The IA-RT test addresses the schedulability testing problem while determining the priorities of the tasks using a multiprocessor extension of the Audsley's optimal priority assignment scheme. Finding such a priority ordering is important since many of the traditional priority-assignment policies (e.g., the deadline-monotonic policy) perform poorly for global FP scheduling of constrained-deadline tasks, and also because the optimal priority assignment for such task systems is not known at present time.

The IA-RT test also deals with the challenge of reducing the pessimism in approximating the worst-case (i.e., critical instant) for global FP scheduling. The IA-RT test is derived based on a crucial observation (regarding the schedulability analysis) which is used to derive an improvement in order to reduce the pessimism in the interference computation as caused by the higher priority tasks on each lower priority task. The observation is that, if a number of m' tasks and m' processors, $0 \leq m' < m$, are not considered during the schedulability analysis of a lower priority task τ_i, then the pessimism of the interference computation due to the higher priority tasks can be reduced. Based on this observation, a novel criterion is proposed which finds a set of m' tasks and m' processors that will not be considered during the global FP schedulability analysis of a lower priority task τ_i. By computing an upper bound on the interference of each lower priority task $\tau_i \in \Gamma$, the response-time-based IA-RT test is derived. The IA-RT test

does not only dominates but also empirically outperforms the state-of-the-art iterative test for global FP scheduling of constrained-deadline sporadic tasks.

C3 Uniprocessor Fault-Tolerant Scheduling (Chapter 7) – A fault-tolerant scheduling algorithm for uniprocessors, called FTDM, based on the DM priority assignment policy is proposed. The proposed scheduling algorithm considers a very general fault model such that multiple faults can occur in any task and at any time (even during recovery). The FTDM algorithm considers time-redundant execution of the tasks as backup to recover from occurrences of maximum f task errors within each of all possible time intervals of length D_{max}. In order to resolve the interdependency between meeting timing constraints and achieving fault-tolerance using time redundancy, precise schedulability analysis of FTDM algorithm is conducted. An exact schedulability test is derived based on the maximum total workload requested within the release time and absolute deadline of the job of each task released at the critical instant. To calculate this maximum total workload, assuming occurrences of multiple faults, a novel technique to compose the execution time of the higher priority jobs is used.

The only work that deals with a similar fault model as the FTDM algorithm is proposed by Aydin [Ayd07], but considers EDF priority and the exact test in [Ayd07] has an exponential run-time complexity. On the other hand, the run time-complexity to evaluate the exact schedulability test of the proposed FTDM algorithm is $O(n \cdot \hat{N} \cdot f^2)$, where $\hat{N}$ is the maximum number of jobs (generated by the n periodic tasks) released within any time interval of length D_{max}. No previous work has derived an exact fault-tolerant uniprocessor schedulability test that has a lower time complexity than that is presented in this thesis for the assumed fault model. The proposed schedulability test can be applied to partitioned multiprocessor scheduling during assignment of the tasks to the processors so that a maximum of f task errors can be tolerated on each processor.

C4 Multiprocessor Fault-Tolerant Scheduling (Chapter 8) – A fault-tolerant FP scheduling algorithm for multiprocessors, called FTGS, based on global scheduling paradigm assuming an arbitrary fixed-priority ordering of the tasks is proposed. The fault model of FTGS algorithm is as general as the FTDM algorithm. In addition, the FTGS algorithm also considers tolerating permanent processor failures in its fault model. More specifically, the FTGS scheduling considers tolerating ρ permanent processor failures within the lifetime of the system, in addition to tolerating a maximum of f task errors that can occur within any interval equal to D_{max}. No other work considers a powerful fault model for multiprocessor scheduling as is assumed for the FTGS algorithm.

The schedulability analysis of the FTGS algorithm does not only resolve the interdependency between timeliness and achieving fault tolerance using time redundancy, but also addresses the priority assignment problem, which is common even for traditional (non-fault-tolerant) global FP scheduling. To that end, a sufficient schedulability test for FTGS scheduling with a time-complexity of $O(n^2 \cdot f^2 \cdot max\{\hat{N}, m \cdot f, f^2\})$ is derived. The schedulability test for the FTGS algorithm can be combined with Audsley's optimal priority assignment algorithm to search for a priority ordering in case the test is not satisfied for the given priority ordering of the tasks.

The mathematical expression of the FTGS schedulability test incorporates different parameters from the system models: f (number of task errors), ρ (number of processor failures) and m (number of available processors) along with the parameters of the task set. The system designers can play around with different values of these parameters to make trade-off between fault resilience and resource requirement of the system. While most of the previous work consider tolerating a task error using techniques intended for tolerating processor failures (a wasteful approach in terms of resources), the FTGS algorithm distinguishes between task errors and processors failures to efficiently utilize the computing resources while at the same time achieving fault-tolerance.

C5 Multiprocessor Scheduling of Mixed-Criticality Systems (Chapter 9) – A certification-cognizant FP multiprocessor scheduling algorithm, called MSM, for constrained-deadline sporadic tasks having different criticality levels is proposed. The proposed MSM scheduling algorithm is based on a global FP scheduling paradigm with an additional feature — runtime monitoring of the criticality behavior — that determines when the system switches to a higher criticality behavior[1]. Upon detection of criticality switch to a higher criticality behavior, tasks relevant only to that criticality behavior are dispatched for execution. The run-time monitoring capability enables the MSM algorithm to address both the deadline and criticality aspects of MC tasks. A sufficient response-time-based schedulability test of the MSM algorithm is proposed. This schedulability test can be used to verify whether the timing constraints of the tasks at each criticality levels are met, or not, thereby facilitating certification.

The main objective for deriving the schedulability test for the MSM scheduling is to make the test applicable with Audsley's OPA algorithm so that the fixed priority assignment of the MC tasks can be determined. Finding such a priority ordering is important because many of the heuristic priority-assignment policies, for example, criticality-as-priority-assignment (CAPA), perform poorly for FP scheduling of mixed criticality tasks. While many other earlier work consider only two different criticality levels, the MSM algorithm considers an arbitrary number of criticality levels (which is important since the tasks in many practical systems have more than two criticality levels). This is the first published work, on global FP scheduling of MC tasks on multiprocessors. Although this work considers FP scheduling, it can be easily extended for any other work-conserving scheduling algorithm. The time complexity to evaluate the schedulability test for MSM algorithm, combined with the OPA algorithm for a task set with $\mathcal{L}$ criticality levels, is $O(n^2 \cdot \mathcal{L} \cdot T^{\mathcal{L}}_{max})$, which is pseudo-polynomial for any fixed value of $\mathcal{L}$ that is reasonable for practical mixed-criticality systems. For example, the time complexity for dual-criticality system (i.e., MC system with only two criticality levels) is $O(n^2 \cdot T^2_{max})$ which is pseudo-polynomial in the representation of the task set. Simulation result shows that the schedulability test for MSM algorithm combined with Audsley's OPA algorithm significantly outperforms the schedulability test for MSM scheduling using other traditional priority assignment (e.g., deadline-monotonic, CAPA) policies.

[1]The criticality behavior of the system at each time instant is determined based on the actual execution time of the active job of each task at that time instant.

5
Density-Bound-Based Test

A new fixed-priority assignment policy, called *Improved Slack-Monotonic Density Separation* (ISM-DS), for global FP scheduling of a set of constrained-deadline sporadic tasks is presented in this chapter. Based on a threshold density, that only depends on the number of processors, the priority assignment policy ISM-DS assigns slack-monotonic priority to a subset of the tasks while each of the other tasks is assigned the highest fixed-priority. A sufficient density-bound-based schedulability test is derived for global FP scheduling where the priorities are assigned according to policy ISM-DS. The derived density-bound test dominates the state-of-the-art density-bound test for global FP scheduling of constrained-deadline sporadic tasks.

Based on the schedulability analysis of priority assignment policy ISM-DS, another priority assignment policy, called ISM-DS$[\xi]$, is proposed. Policy ISM-DS$[\xi]$ assigns the priorities similar to policy ISM-DS except that the threshold density ξ is searched from the set of densities of all the tasks. Considering the schedulability testing problem, the aim for searching the threshold density ξ is to guarantee the schedulability of the tasks for the ISM-DS$[\xi]$ priority assignment policy. It is proved that the schedulability test of global FP scheduling using ISM-DS$[\xi]$ as the priority assignment policy dominates that of using the ISM-DS priority assignment policy. Empirical investigation using randomly generated task sets shows surprising improvement of the schedulability test for policy ISM-DS$[\xi]$ over that of using policy ISM-DS.

5.1 Introduction

It has become obvious that continuously increasing the clock speeds of uniprocessors to provide more performance is impossible due to power consumption and heat dissipation limits. The processor industry has adopted multicore architectures to provide the growing demand of computation power. While real-time scheduling of sporadic tasks on uniprocessors is considered to be mature enough, real-time scheduling theory for multiprocessors is still young and has recently received considerable attention.

The main design goal of many global [ABJ01, Bak06, BCL05, And08a, BCL09, BC07, GSYY09, DB11b] and partitioned [DL78, LBOS95, LDG04, AJ03, FBB06, LMM98a, LGDG03, OB98, OS95b] fixed-priority scheduling algorithms is to derive a *schedulability test* that when satisfied implies that all the deadlines are met. The global scheduling approach is being seriously considered for many practical systems since different techniques, e.g., inter-core prefetching [KST11], locked-cache [SMR11], push-assisted migration [SMRM09], have been proposed to reduce the overhead due to migration. The FP scheduling policy is the preferred scheduling policy in the industry due to its flexibility, ease of implementation and debugging [ABB96, SG90, SLR86, XP00, AS06]. Almost all commercial real-time kernel / operating systems (e.g. VxWorks, RT-Linux, RT-Mach), languages (e.g. Ada95) support fixed-priority scheduling. These observations motivate the design and analysis of global FP scheduling algorithms in this thesis. The following real-time scheduling problem is addressed in this chapter:

> **Given a collection of n constrained-deadline sporadic tasks, is it possible to meet all the task deadlines when the tasks are FP scheduled on m identical, unit-capacity processors?**

Challenges. As already pointed out in Chapter 1 that there are two major research challenges in the context of global FP scheduling: (i) *priority assignment problem*, and (ii) *schedulability testing problem.* The optimal FP ordering for constrained-deadline tasks scheduled on uniprocessors is known [LW82]: deadline-monotonic priority ordering is the optimal FP ordering in such case. However, the optimal FP ordering of global multiprocessor scheduling of constrained-deadline tasks is still unknown [DB11a]. Moreover, it has already been shown by Dhall and Liu [DL78] that the utilization bound of global FP scheduling of implicit-deadline task based on rate-monotonic priority ordering is 0%. This result can easily be extended to show that the density bound of global FP scheduling of constrained-deadline tasks according to deadline-monotonic priority assignment policy is also 0%. To achieve higher utilization/density bound, researchers have proposed new fixed-priority assignment policy with non-zero utilization/density bound [ABJ01, Bak06, BCL05, And08a, Lun02].

Deriving an effective schedulability test is equally important as deriving a "good" fixed-priority ordering since hard real-time system needs to apply schedulability test before the system is in mission. The challenge during the schedulability analysis of global FP scheduling in order to derive a schedulability test involves correctly predicting the worst-case runtime behavior and analyzing this worst-case behavior. Unfortunately, the

worst-case (known as critical instant, see Section 3.1) for global FP scheduling of sporadic tasks is not known [LMM98b]. However, several interesting schedulability analysis techniques have been proposed by researchers to analyze global FP scheduling to derive sufficient schedulability test. The amount of pessimism used during such schedulability analysis determines the utilization/density bound for different fixed-priority assignment policies proposed in [ABJ01, Lun02, BCL05, Bak06, And08a].

Contributions. One of the most expressive ways to derive a schedulability test for implicit- and constrained-deadline tasks is in terms of its *utilization bound* and *density bound*, respectively. It has already been proved that neither global nor partitioned FP scheduling can have a utilization bound greater than $0.5m$ on m processors for implicit-deadline task systems [ABJ01, CFH$^+$04]. There exists a partitioned FP scheduling algorithm, called R-BOUND-MP-NFR, having utilization bound of $0.5m$ [AJ03]. However, the state-of-the-art utilization bound of global FP scheduling is $\frac{m+1}{3}$ for $m \leq 6$ (RM-US[$\frac{1}{3}$] scheduling [BCL05]) and $\frac{2m}{3+\sqrt{5}}$ for $m > 6$ (SM-US[$\frac{2}{3+\sqrt{5}}$] scheduling [And08a]) for implicit-deadline sporadic task systems. The state-of-the-art density bound of global FP multiprocessor scheduling of constrained-deadline tasks is $\frac{m+1}{3}$ where priorities are assigned based on DM-DS[$\frac{1}{3}$] priority assignment policy [BCL05].

This chapter presents a new priority assignment policy, called ISM-DS, and derives a corresponding density bound for global FP scheduling. It is also proved that the density bound of global FP scheduling using policy ISM-DS is higher than that of DM-DS[$\frac{1}{3}$] for constrained-deadline task sets. The density bound of the proposed priority assignment policy ISM-DS becomes the utilization bound for implicit-deadline task sets. It will be shown that the utilization bound using priority assignment policy ISM-DS is higher than that of both RM-US[$\frac{1}{3}$] and SM-US[$\frac{2}{3+\sqrt{5}}$] for implicit-deadline task systems for any finite $m \geq 2$.

The ISM-DS priority assignment policy assigns priorities to the tasks based on some threshold density: each task having density greater than the threshold density is assigned the highest fixed-priority and the remaining tasks are assigned lower, slack-monotonic priorities. The threshold density for policy ISM-DS depends only on the number of processors and does not consider the parameters (e.g. density) of the tasks in a task set. By considering density of the tasks in addition to the number of processors, the threshold density can be searched from the set of densities of all the tasks. To this end, another priority assignment policy, called ISM-DS[ξ], is proposed where the threshold density ξ is searched from the set of densities of the tasks in a given task set. If such a threshold density ξ can be found, then the task set is schedulable using global FP scheduling based on priority assignment policy ISM-DS[ξ]. It is shown that, the schedulability test for global FP scheduling using priority assignment policy ISM-DS[ξ] dominates that of the density-bound test derived for ISM-DS policy.

Organization. Section 5.2 presents related work. Then, some important parameters of the task model is presented in Section 5.3. The priority assignment policy ISM-DS and its corresponding density bound for global FP scheduling of constrained-deadline sporadic tasks is proposed in Section 5.4. Then, the priority assignment policy ISM-DS[ξ]

is proposed in Section 5.5. Empirical investigation using randomly generated task sets to compare the derived schedulability tests for priority assignment policy ISM-DS[ξ] and ISM-DS is presented in Section 5.6. Then, a utilization based test for implicit-deadline tasks based on priority assignment policy ISM-DS is presented in Section 5.7. The schedulability analysis of global FP scheduling using ISM-DS priority assignment policy enables the derivation of a utilization bound for uniprocessor slack-monotonic scheduling in Section 5.8. Finally, Section 5.9 summarizes this chapter.

5.2 Related Work

While the well-known RM priority assignment is optimal for uniprocessor FP scheduling of implicit-deadline tasks [LL73], it is is not optimal for global FP scheduling on multiprocessors due to so called the "Dhall's effect" [DL78]. Dhall and Liu showed that global multiprocessor scheduling of implicit-deadline tasks under RM priority assignment has system utilization 0% as $m \rightarrow \infty$. The problem due to Dhall's effect is the existence of a task with high utilization but having a relatively lower RM priority.

In order to circumvent Dhall's effect, many of the work around global scheduling have considered intelligent fixed-priority assignment policy based on hybrid-priority assignment (HPA) scheme. In HPA scheme, each task in a subset of the tasks is given the highest fixed priorities while the remaining tasks are assigned some other, lower fixed priorities. The HPA policy has been used in the development of numerous global FP scheduling algorithms and their corresponding schedulability tests, the first being the RM-US[$\frac{m}{3m-2}$] algorithm proposed by Andersson, Baruah and Jonsson [ABJ01]. That algorithm was shown to have a utilization bound of $\frac{m^2}{3m-2}$ on m processors for implicit-deadline tasks. The RM-US[$\frac{m}{3m-2}$] algorithm manages to avoid the Dhall's effect by assigning the highest fixed priority to the tasks having utilization greater than $\frac{m}{3m-2}$ while the rest of the tasks are assigned priorities according to the traditional RM policy. Lundberg [Lun02] later showed that using RM hybrid priority assignment scheme, RM-US can achieve a utilization bound of approximately $0.374m$.

In [Bak06], Baker presented an analysis of global FP scheduling. Baker's analysis is general for any fixed-priority scheduling and arbitrary-deadline task systems. Based on a derivation of the minimum amount of interference in an interval that can cause a task's deadline to be missed, Baker showed that, for implicit-deadline sporadic task sets, the utilization bound of RM scheduling is $\frac{m(1-u_{max})}{2} + u_{min}$, where u_{max} and u_{min} are the maximum and minimum utilization of any task in the task set, respectively. The RM scheduling is studied for uniform multiprocessors (i.e., processors having different speeds) by Baruah and Goossens in [BG03a], and it is shown that the utilization bound is $\frac{m}{3}$ for implicit-deadline tasks on m unit-capacity processors if no task has utilization greater than $1/3$.

Bertogna et al. [BCL05] proposed an algorithm, called RM-US[$\frac{1}{3}$], which is an improvement of the algorithm RM-US[$\frac{m}{3m-2}$] in [ABJ01] for implicit-deadline sporadic task systems. Based on schedulability analysis of the deadline-monotonic pri-

ority assignment, Bertogna et al. proved that the utilization bound of the a HPA-based RM-US[$\frac{1}{3}$] algorithm is $\frac{m+1}{3}$ for implicit-deadline tasks. The RM-US[$\frac{1}{3}$] algorithm assigns the highest priority to the tasks having utilization greater than $1/3$ while the rest of the tasks are given the traditional RM priority. The authors also showed that if the total density of a constrained-deadline task set is not greater than $\frac{m+1}{3}$ (i.e., density-bound), then all deadlines are met using DM-DS[$\frac{1}{3}$] priority assignment policy. According to DM-DS[$\frac{1}{3}$], if a task's density is greater than $\frac{1}{3}$, then it is given the highest fixed-priority, otherwise, it is given the traditional DM priority.

Andersson [And08a] proposed the SM-US[$\frac{2}{3+\sqrt{5}}$] priority assignment policy based on a slack-monotonic HPA scheme that has a utilization bound of $\frac{2m}{3+\sqrt{5}}$ for global FP scheduling of implicit-deadline sporadic task systems. According to SM-US[$\frac{2}{3+\sqrt{5}}$], each task having utilization greater than $\frac{2}{3+\sqrt{5}}$ is given the highest fixed priority while the rest of the tasks are assigned slack-monotonic priorities.

The state-of-the-art utilization bound for global FP multiprocessor scheduling of implicit-deadline sporadic tasks is $\frac{m+1}{3}$ for $m \leq 6$ (RM-US[$\frac{1}{3}$] scheduling [BCL05]) and $\frac{2m}{3+\sqrt{5}}$ for $m > 6$ (SM-US[$\frac{2}{3+\sqrt{5}}$] scheduling [And08a]). The state-of-the-art density bound for global FP multiprocessor scheduling of constrained-deadline sporadic tasks is $\frac{m+1}{3}$ (DM-DS[$\frac{1}{3}$] scheduling [BCL05]). In this thesis, a new slack-monotonic HPA policy, called ISM-DS, for constrained-deadline sporadic task sets is proposed. It is proved that the density bound for global FP scheduling of constrained-deadline sporadic tasks using policy ISM-DS is $m \cdot min\{\frac{1}{2}, \frac{3m-2-\sqrt{5m^2-8m+4}}{2m-2}\}$, which is higher than the density bound of DM-DS[$\frac{1}{3}$] scheduling for constrained-deadline sporadic task sets. The density bound of global FP scheduling using policy ISM-DS becomes the utilization bound for implicit-deadline task sets. The bound $m \cdot min\{\frac{1}{2}, \frac{3m-2-\sqrt{5m^2-8m+4}}{2m-2}\}$ for global FP scheduling of implicit-deadline task systems is higher than that of both the RM-US[$\frac{1}{3}$] and SM-US[$\frac{2}{3+\sqrt{5}}$] scheduling for any finite $m \geq 2$.

5.3 Parameters of Task Model

The task model considered in this chapter is constrained-deadline sporadic task system where each task $\tau_i \in \Gamma$ is characterized by a triple (C_i, D_i, T_i). Please see Section 3.1 (page 33) for details of the task model.

The *slack* of each task τ_i is defined to be equal to $(D_i - C_i)$. Note that slack of an implicit-deadline task τ_i is $(T_i - C_i)$. Task τ_i has higher Slack-Monotonic (SM) priority than task τ_j only if the following condition[1] is satisfied:

$$(D_i - C_i) < (D_j - C_j)$$

Without any loss of generality, the tasks in set Γ are assumed to be sorted based on decreasing priority order (i.e., τ_1 is the highest priority task and τ_n is the lowest priority

[1] Ties, i.e., $(D_i - C_i) = (D_j - C_j)$, can be broken arbitrarily.

task). For a given priority ordering of the tasks, the execution of a task τ_k can only be interfered by the higher-priority tasks in global FP scheduling. In other words, whether task τ_k meets its deadline or not depends only on the tasks in set $\mathsf{HP}_k \cup \{\tau_k\}$. The task set Γ^k is defined as follows:

$$\Gamma^k \stackrel{\text{def}}{=} \mathsf{HP}_k \cup \{\tau_k\}$$

where τ_k is the lowest priority task in Γ^k and $\mathsf{HP}_k = \{\tau_1, \ldots, \tau_{k-1}\}$ for $k = 1, 2 \ldots n$. Note that $\Gamma^j \subseteq \Gamma^k$ for $1 \leq j \leq k \leq n$. The *total density* δ^k_{sum} of the task set Γ^k is defined as follows:

$$\delta^k_{sum} = \sum_{\tau_i \in \Gamma^k} \delta_i = \sum_{\tau_i \in \Gamma^k} \frac{C_i}{D_i}$$

for $k = 1, 2 \ldots n$. The *maximum density* and *minimum density* of a sporadic task system Γ^k are denoted respectively as δ^k_{max} and δ^k_{min} such that $\delta^k_{min} \leq \delta_i \leq \delta^k_{max}$ for all $\tau_i \in \Gamma^k$. Formally,

$$\delta^k_{max} = \max_{\tau_i \in \Gamma^k} \{\delta_i\}$$

$$\delta^k_{min} = \min_{\tau_i \in \Gamma^k} \{\delta_i\}$$

The *total utilization* $U^k$of the task set Γ^k is given as follows:

$$U^k = \sum_{\tau_i \in \Gamma^k} u_i = \sum_{\tau_i \in \Gamma^k} \frac{C_i}{T_i}$$

for $k = 1, 2 \ldots n$. The *maximum density* and *minimum density* of a sporadic task system Γ^k are denoted respectively as u^k_{max} and u^k_{min} such that $u^k_{min} \leq \delta_i \leq u^k_{max}$ for all $\tau_i \in \Gamma^k$. Formally,

$$u^k_{max} = \max_{\tau_i \in \Gamma^k} \{u_i\}$$

$$u^k_{min} = \min_{\tau_i \in \Gamma^k} \{u_i\}$$

5.4 Constrained-Deadline Tasks: Density-Bound

In this section, the priority assignment policy `ISM-DS` and a corresponding density-bound-based schedulability test for constrained-deadline task systems are presented. The proposed priority-assignment policy `ISM-DS` is based on a *slack-monotonic* HPA policy that works as follows: if the density of a task is not greater than a *threshold density*, say δ_{ts}, then the task is assigned a priority according to the slack-monotonic policy; otherwise, the task is given the highest fixed priority.

The main challenge for such HPA policy is to find the threshold density δ_{ts} which determines the two subsets of the task set such that tasks in one subset are given the slack-monotonic priorities and each of the tasks in the other subset is given the highest fixed-priority, where ties are broken arbitrarily at runtime. The threshold density δ_{ts} for

policy ISM-DS is determined based on the schedulability analysis of a class of task sets, called "special" task sets. A task set is said to be "special on m processors" based on *two* particular properties (defined shortly in subsection 5.4.2).

The threshold density used for policy ISM-DS is $\frac{3m-2-\sqrt{5m^2-8m+4}}{2m-2}$ where m is the number of processors, $m \geq 2$. Thus, given the number of processors m, the threshold density for ISM-DS is computed and all the tasks are assigned the fixed priorities according to the slack-monotonic HPA policy. It is proved that the density bound of global FP scheduling of constrained-deadline sporadic tasks using policy ISM-DS is $m \cdot min\{\frac{1}{2}, \frac{3m-2-\sqrt{5m^2-8m+4}}{2m-2}\}$. It is easy to see that this density bound is larger than that of the state-of-the-art DM-DS$[\frac{1}{3}]$ scheduling proposed in [BCL05].

The proof strategy to derive the density bound is as follows. First, it will be shown that a "special" task set Γ^k (which is a subset of the original task set Γ) is schedulable by global FP scheduling based on slack-monotonic priority assignment (subsection 5.4.2). Second, two general conditions are derived when satisfied imply that the entire task set Γ is schedulable using a slack-monotonic HPA policy using some threshold density δ_{ts} (subsection 5.4.3). Finally, the value of the threshold density δ_{ts} for policy ISM-DS and the corresponding density bound for global FP scheduling of the entire task set is derived (subsection 5.4.4). The following results and definitions in subsection 5.4.1 will be used in the remainder of this section.

5.4.1 Prior Results and Useful Definitions

When analyzing the schedulability of a lower priority task τ_j using any global FP scheduling within the interval $[t_1, t_2)$, its schedulability depends on the amount of work done by the higher priority tasks within $[t_1, t_2)$. By assuming that a job of an implicit-deadline sporadic task τ_j arrives at t_1 and misses its deadline (which is the first missed deadline in the schedule) at t_2 such that $t_2 = t_1 + T_j$, the analysis by Andersson in [And08a] proved that the maximum amount of execution required within $[t_1, t_2)$ by a higher priority task $\tau_i \in \mathsf{HP}_j$ is $C_i + (T_j - C_i)\frac{C_i}{T_i}$, whenever $u^j_{max} \leq \frac{m}{2m-1}$ and $U^j \leq \frac{m^2}{2m-1}$. This result by Andersson [And08a] is given in Lemma 5.1.

Lemma 5.1 (Based on [And08a])**.** *Consider global* FP *scheduling of an implicit-deadline sporadic task system Γ^j on m processors by assuming that $u^j_{max} \leq \frac{m}{2m-1}$, $U^j \leq \frac{m^2}{2m-1}$ and that all the tasks in HP_j are schedulable. When analyzing the schedulability of the lowest priority task τ_j within $[t_1, t_2)$, the maximum amount of execution by all the higher priority tasks during $[t_1, t_2)$ is at most:*

$$\sum_{\tau_i \in \mathsf{HP}_j} C_i + (L - C_i)\frac{C_i}{T_i} \tag{5.1}$$

where $L = t_2 - t_1 = T_j$.

Proof. Eq. (5.1) is derived by Andersson in [And08a] (please see Eq. (16) in reference [And08a] for this derivation). □

By considering the constrained relative deadline instead of implicit relative deadline and considering density instead of utilization, the proof and result of Lemma 5.1 are directly applicable to constrained-deadline task systems. Corollary 5.2 is the adaptation of Lemma 5.1 for constrained-deadline task systems and will be used later in this section to upper bound the work of the tasks in HP_j within an interval $[t_1, t_2)$.

Corollary 5.2 (Based on Lemma 5.1)**.** *Consider global* FP *scheduling of a constrained-deadline sporadic task system Γ^j on m processors by assuming that $\delta^j_{max} \leq \frac{m}{2m-1}$, $\delta^j_{sum} \leq \frac{m^2}{2m-1}$ and that all the tasks in HP_j are schedulable. When analyzing the schedulability of the lowest priority task τ_j within $[t_1, t_2)$, the maximum amount of execution by all the higher priority tasks during $[t_1, t_2)$ is at most:*

$$\sum_{\tau_i \in \mathsf{HP}_j} C_i + (L - C_i)\frac{C_i}{D_i} \tag{5.2}$$

where $L = t_2 - t_1 = D_j$.

Proof. Eq. (5.2) can be derived similar to the derivation of Eq. (5.1) by considering constrained relative deadline instead of implicit relative deadline. □

Function $F_m(x)$: The following function in Eq. (5.3) is used in the remainder of this chapter:

$$F_m(x) \ = \frac{m(1-x)}{2-x} + x \tag{5.3}$$

where $m \in \mathbb{Z}^+$ and $0 \leq x \leq \frac{m}{2m-1}$. Two important features of the function in Eq. (5.3) are given in Lemma 5.3.

Lemma 5.3. *Consider a, b, x, c and d such that $0 \leq a \leq b \leq x \leq c \leq d \leq \frac{m}{2m-1}$ for any integer $m > 0$. The following two inequalities hold:*

$$min\{F_m(b), F_m(c)\} \leq F_m(x) \tag{5.4}$$

$$min\{F_m(a), F_m(d)\} \leq min\{F_m(b), F_m(c)\} \tag{5.5}$$

Proof. Proof is given in Appendix A (page 219). □

Corollary 5.2 and Lemma 5.3 are used in the remainder of this chapter. The global FP schedulability analysis of task set Γ presented in this section is based on the schedulability analysis of a class of task sets called "special" task sets. A task set is said to be "special on m processors" based on *two* particular properties defined in Definition 5.1. It will be shown in Theorem 5.1 that a task set that is special on m processors is schedulable using global slack-monotonic scheduling, denoted by $\mathsf{GS}_{\mathsf{SM}}$, on m processors. The $\mathsf{GS}_{\mathsf{SM}}$ scheduling is global FP scheduling where all the tasks are assigned fixed priorities based on slack-monotonic priority assignment policy.

5.4.2 "Special" Task Set and its Schedulability

In this subsection, the two properties of a sporadic task system Γ^k that is "special" on m processors are formally presented. It will be proved that all the deadlines of the special task system Γ^k are met using algorithm $\mathsf{GS_{SM}}$ on m processors.

Definition 5.1 (Special Task System)**.** *A constrained-deadline sporadic task system Γ^k is special on m processor if it satisfies the following two properties:*

$$\textbf{Property 1: } \delta^k_{max} \leq \frac{m}{2m-1}$$

$$\textbf{Property 2: } \delta^k_{sum} \leq min\{F_m(\delta^k_{min})\ ,\ F_m(\delta^k_{max})\}$$

According to Property 1, the maximum density of any task in Γ^k, that is special on m processors, is not greater than $\frac{m}{2m-1}$. According to Property 2, the total density of the special task system Γ^k is not greater than the minimum of $F_m(\delta^k_{min})$ and $F_m(\delta^k_{max})$. Before the global slack-monotonic schedulability analysis of a special task set Γ^k is presented, the following Lemma 5.4 (proof is in Appendix A, page 220) is required.

Lemma 5.4. *Consider sporadic task system Γ^k that is special on m processors. The following inequality holds for $m \geq 1$*

$$min\{F_m(\delta^k_{min})\ ,\ F_m(\delta^k_{max})\}\ \leq\ \frac{m^2}{2m-1} \tag{5.6}$$

Slack-Monotonic Global Schedulability Analysis of Special Task System

It will be proved that a sporadic task system Γ^k that is special on m processors is schedulable using $\mathsf{GS_{SM}}$ on m processors. First, by assuming that all the tasks in HP_j meet their deadlines, it is shown in Lemma 5.5 that all the jobs of the lowest priority task τ_j of task set Γ^j, which is special on m processors, complete by their deadlines using $\mathsf{GS_{SM}}$ scheduling of Γ^j on m processors. Then, by inductively applying Lemma 5.5 on special task set Γ^j for $j = 1, 2, \ldots k$, it is proved that special task system Γ^k is also schedulable on m processors using global scheduling algorithm $\mathsf{GS_{SM}}$.

Lemma 5.5. *Consider sporadic task set Γ^j that is special on m processors. If all the tasks in HP_j meet deadlines using $\mathsf{GS_{SM}}$ on m processors, then all the jobs of task τ_j also meet their deadlines when $\Gamma^j = \mathsf{HP}_j \cup \{\tau_j\}$ is scheduled using $\mathsf{GS_{SM}}$ on m processors.*

Proof. This Lemma is proved using induction. Let's assume that all the $(l-1)$ jobs of τ_j have met their deadlines using $\mathsf{GS_{SM}}$ scheduling algorithm. It will be proved that the l^{th} job of τ_j also meets the deadline. Using induction on $l \geq 1$, the correctness of Lemma 5.5 then immediately follows. For a special task set Γ_j, we have $\delta^j_{max} \leq \frac{m}{2m-1}$ (from Property 1 of Definition 5.1) and $\delta^j_{sum} \leq \frac{m^2}{2m-1}$ (from Property 2 of Definition 5.1 and Eq. (5.6) of Lemma 5.4). Remember that all the tasks in HP_j are schedulable using $\mathsf{GS_{SM}}$ on m processors (premise of this lemma).

Let the l^{th} job of task τ_j be released at time r. This job requires C_j units of execution time before its deadline $(r + D_j)$. Therefore, when considering the schedulability of the l^{th} job within the interval $[r, r + D_j)$, Corollary 5.2 can be applied by setting $L = (r + D_j) - r = D_j$. And, according to Eq. (5.2) of Corollary 5.2, the maximum amount of execution required by the higher priority tasks in HP_j during $[r, r + D_j)$ is at most:

$$\sum_{\tau_i \in \mathsf{HP}_j} C_i + (D_j - C_i)\frac{C_i}{D_i} \tag{5.7}$$

The amount of processor capacity left unused by the tasks in HP_j during the interval $[r, r + D_j)$ on m processors is therefore at least

$$m \cdot D_j - \sum_{\tau_i \in \mathsf{HP}_j} (C_i \ + (D_j - C_i\ )\delta_i) \tag{5.8}$$

In the worst case (i.e., all the m processors are available at the same time) $\frac{1}{m}$ of this unused capacity can be used by τ_j. Consequently, the amount of processing capacity available to the l^{th} job of τ_j during the interval $[r, r + D_j)$ on m processors is at least

$$\frac{1}{m}\left[m \cdot D_j - \sum_{\tau_i \in \mathsf{HP}_j} (C_i \ + (D_j - C_i\ )\delta_i)\right]$$

To guarantee that the l^{th} job of τ_j meets its deadline, this capacity needs to be at least as large as the execution time of τ_j; that is, we must have,

$$C_j \leq \frac{1}{m}\left[m \cdot D_j - \sum_{\tau_i \in \mathsf{HP}_j} (C_i \ + (D_j - C_i\ )\delta_i)\right] \tag{5.9}$$

In the remaining part of this proof, it is shown that Eq. (5.9) holds; which guarantees that the l^{th} job of τ_j meets its deadline. Since task set Γ^j is special on m processors, according to Property 2 of special task set we have

$$\delta^j_{sum} \leq \mathit{min}\{F_m(\delta^j_{min})\ , F_m(\delta^j_{max})\} \tag{5.10}$$

For task $\tau_j \in \Gamma^j$, we have $\delta^j_{min} \leq \delta_j \leq \delta^j_{max}$. Thus, according to Property 1 of special task system Γ^j, we also have $0 \leq \delta^j_{min} \leq \delta_j \leq \delta^j_{max} \leq \frac{m}{2m-1}$. And using Eq. (5.4) of Lemma 5.3, it follows that

$$\mathit{min}\{F_m(\delta^j_{min})\ , F_m(\delta^j_{max})\} \leq F_m(\delta_j) \tag{5.11}$$

From Eq. (5.10) and Eq. (5.11), we have $\delta^j_{sum} \leq F_m(\delta_j)$ which is equivalent to

$$
\begin{aligned}
&\equiv \sum_{\tau_i \in \mathsf{HP}_j \cup \{\tau_j\}} \delta_i \leq \frac{m(1-\delta_j)}{2-\delta_j} + \delta_j \qquad \text{[from Eq. (5.3)]} \\
&\equiv \sum_{\tau_i \in \mathsf{HP}_j} \delta_i \leq \frac{m(1-\delta_j)}{2-\delta_j} \\
&\equiv \sum_{\tau_i \in \mathsf{HP}_j} \delta_i(2-\delta_j) \leq m(1-\delta_j) \\
&\equiv \delta_j \leq 1 - \frac{1}{m} \sum_{\tau_i \in \mathsf{HP}_j} \delta_i(2-\delta_j) \\
&\equiv \delta_j \leq \frac{1}{m}\Big[m - \sum_{\tau_i \in \mathsf{HP}_j} \big[\delta_i + \delta_i(1-\delta_j)\big]\Big] \\
&\equiv \frac{C_j}{D_j} \leq \frac{1}{m}\Big[m - \sum_{\tau_i \in \mathsf{HP}_j} [\frac{C_i}{D_i} + \frac{C_i}{D_i}(\frac{D_j - C_j}{D_j})]\Big]
\end{aligned}
$$

$$
\begin{aligned}
\Rightarrow\ & (\text{According to slack-monotonic priorities} \\
& \quad \forall i \in \mathsf{HP}_j\colon\ (D_i - C_i) \leq (D_j - C_j)) \\
& \frac{C_j}{D_j} \leq \frac{1}{m}\Big[m - \sum_{\tau_i \in \mathsf{HP}_j} [\frac{C_i}{D_i} + \frac{C_i}{D_i}(\frac{D_i - C_i}{D_j})]\Big] \\
\equiv\ & C_j \leq \frac{1}{m}\Big[m \cdot D_j - \sum_{\tau_i \in \mathsf{HP}_j} [\frac{C_i D_j}{D_i} + C_i - \frac{C_i^2}{D_i}]\Big] \\
\equiv\ & C_j \leq \frac{1}{m}\Big[m \cdot D_j - \sum_{\tau_i \in \mathsf{HP}_j} \big[C_i + (D_j - C_i)\delta_i\big]\Big] \ \equiv \text{ Eq. (5.9)}
\end{aligned}
$$

Since the inequality in Eq. (5.9) is true, it can be concluded that the l^{th} job of task τ_j meets its deadline using $\mathsf{GS}_{\mathsf{SM}}$. □

Based on Lemma 5.5, now it will be proved in Theorem 5.1 that the constrained-deadline sporadic task set Γ^k that is special on m processors is schedulable using $\mathsf{GS}_{\mathsf{SM}}$ on m processors.

Theorem 5.1. *A constrained-deadline sporadic task system Γ^k that is special on total m processors is schedulable using $\mathsf{GS}_{\mathsf{SM}}$ scheduling on m processors.*

Proof. Remember that $\Gamma^j \subseteq \Gamma^k$ for $j \leq k$. Thus, it follows that $\delta^j_{sum} \leq \delta^k_{sum}$ and $\delta^j_{max} \leq \delta^k_{max}$. Therefore, from Property 1 and Property 2 of special task set in Definition 5.1, it is evident that if Γ^k is special on m processors, then Γ^j is also special on m

processors for $j \leq k$. Therefore, using induction on $j = 1, 2, \ldots k$ and applying Lemma 5.5 to special task set Γ^j, it is easy to see that the special task system Γ^k is schedulable on m processors using $\mathtt{GS_{SM}}$ scheduling. □

According to Theorem 5.1, a special task set is schedulable using $\mathtt{GS_{SM}}$ algorithm on m processors. The ultimate objective is to find the threshold density for slack-monotonic HPA policy for an arbitrary task set to be scheduled on m processors based on global FP scheduling algorithm. Two general conditions that can imply the global FP schedulability of an arbitrary constrained-deadline sporadic task set Γ for slack-monotonic HPA policy `ISM-US`, based on some threshold density δ_{ts}, are now proposed.

5.4.3 Slack-Monotonic Hybrid Priority Assignment

According to the slack-monotonic HPA policy `ISM-DS`, the priorities to the tasks are assigned based on some threshold density δ_{ts} such that each of the tasks having density not greater than δ_{ts} are given the slack-monotonic priorities and each task having density greater than δ_{ts} is given the highest fixed priority. Using such hybrid policy, the sporadic task set Γ is visualized as the union of two sets $\Gamma = \Gamma_L \cup \Gamma_H$ such that the tasks in set Γ_L have the slack-monotonic priorities and each task in set Γ_H has the highest fixed priority[2]. No task in set Γ_L has higher priority than that of any task in set Γ_H.

The main challenge for slack-monotonic HPA policy is to find the value of δ_{ts} to determine the sets Γ_L and Γ_H. It will be evident shortly that the value of δ_{ts} for priority assignment policy `ISM-DS` depends only on the number of processors. Before the threshold density δ_{ts} for the priority assignment policy `ISM-DS` is determined, two general conditions, denoted as **C1** and **C2**, in Lemma 5.6 that can imply the schedulability of a task set based on HPA-based priority assignment policy `ISM-DS` are presented. The proof strategy in Lemma 5.6 is based on the notion of *predictable scheduling algorithm* proposed by Ha and Liu in [HL94] and used in [And08a] as follows.

Predictability (from [HL94, And08a]): A job is characterized by its arrival time, its deadline, its minimum execution time and its maximum execution time. The execution time of a job is unknown but it is no less than and greater than its minimum and maximum execution time, respectively. A scheduling algorithm A is *predictable* if for every set J of jobs, the following fact

> scheduling all jobs in J by A with execution times equal to their maximum execution times causes all the deadlines to be met

implies that

> scheduling all jobs in J by A with execution times being within at least their minimum execution times and at most their maximum execution times causes all the deadlines to be met.

[2]The subscripts 'L' and 'H' are used to refer to light and heavy tasks, respectively.

This notion of predictable scheduling algorithm implies that it is only needed to analyze the schedulability of the jobs considering the WCET of the jobs. Since a sporadic task set generates a set of jobs, the notion of predictability can be extended in a straightforward manner to algorithms for scheduling sporadic task systems. Ha and Liu's work also implies that global static-priority scheduling of sporadic tasks on multiprocessors is predictable [And08a].

Lemma 5.6. *Let δ_{ts} be the threshold density that is used to determine the sets Γ_L and Γ_H such that $\Gamma = \Gamma_L \cup \Gamma_H$ for the HPA policy* `ISM-DS`*. The sporadic task set Γ is schedulable using global* FP *scheduling if the following two conditions* **C1** *and* **C2** *are satisfied*

(**C1**) $|\Gamma_H| < m$

(**C2**) Γ_L *is special on* $(m - |\Gamma_H|)$ *processors*

Proof. It will be shown that if conditions **C1** and **C2** are true for HPA policy `ISM-DS` that uses δ_{ts} as the threshold density, then the task set Γ is schedulable using global FP scheduling. Consider the following task set Γ'_H such that

$$\Gamma'_H = \{\tau'_i \mid \tau_i \in \Gamma_H,\ C'_i = D_i,\ D'_i = D_i \text{ and } T'_i = T_i\}$$

Note that each task $\tau'_i \in \Gamma'_H$ has density 1 and $|\Gamma'_H| = |\Gamma_H|$. We let $k = |\Gamma'_H| = |\Gamma_H|$.

Now consider the task set $\Gamma' = \Gamma_L \cup \Gamma'_H$ that is to be scheduled on m processors using global FP scheduling where `ISM-DS` is used for priority assignment. According to policy `ISM-DS` that uses the threshold density δ_{ts}, each of the tasks in Γ'_H is given the highest priority and the tasks in Γ_L are given the slack-monotonic priorities.

When scheduling the task set Γ', then at most $k = |\Gamma'_H|$ processors are busy to execute the tasks in set Γ'_H at any time instant since these are the highest priority tasks each with density 1. All these tasks in Γ'_H are schedulable on k processors (one task will get one processor whenever it arrives) since $|\Gamma'_H| = |\Gamma_H| = k < m$ according to **C1**. Therefore, the number of processors that are *always* available for executing the tasks in set Γ_L is at least $(m - k) = (m - |\Gamma_H|)$.

According to **C2**, the tasks in set Γ_L are special on $(m - |\Gamma_H|)$ processors. Since $|\Gamma_H| = k$ and at least $(m - k)$ processor are always available for executing the tasks in set Γ_L, the task set Γ_L is schedulable using $\mathtt{GS_{SM}}$ on $(m - k)$ processors according to Theorem 5.1. Consequently, the task set Γ' is schedulable on total m processors using global FP scheduling where priorities are assigned based on policy `ISM-DS` if conditions **C1** and **C2** are satisfied.

The predictability of global FP scheduling has the following consequence: if the jobs of a task τ_i in a constrained-deadline task set are schedulable using global FP scheduling algorithm A on m processors considering WCET equal to C'_i such that $C'_i = D_i$, then the jobs of τ_i are also schedulable considering its WCET equal to C_i using algorithm A on m processors. Since the jobs of the tasks in $\Gamma' = \Gamma_L \cup \Gamma'_H$ (where each task $\tau'_i \in \Gamma'_H$ has $C'_i = D_i$) is global FP schedulable on m processors using priority assignment

policy ISM-DS, the predictability of global FP scheduling implies that the jobs of the tasks in $\Gamma = \Gamma_L \cup \Gamma_H$ are also global FP scheduling using priority assignment policy ISM-DS whenever **C1** and **C2** are true. □

Guided by the two conditions (**C1** and **C2**) of Lemma 5.6, the following general and an important observation regarding the HPA policy can be made.

Observation 5.1. *The HPA policy can guarantee the schedulability of a task set using global* FP *scheduling if k tasks are given the highest fixed priority and the remaining $(n-k)$ tasks are global* FP *schedulable on at most $(m-k)$ processors using some other fixed-priority assignment, for some k, $0 \leq k < m$.*

This observation will be used in this and other chapters. Now, based on the two general conditions (**C1** and **C2**) of Lemma 5.6, the threshold density δ_{ts} for priority assignment policy ISM-DS and its corresponding density bound of global FP scheduling of an constrained-deadline sporadic task set Γ is presented in subsection 5.4.4.

5.4.4 Density Bound for Policy ISM-DS

In this section, the threshold density used for ISM-DS priority assignment policy is proposed and the corresponding density bound for global FP scheduling of constrained-deadline sporadic tasks is derived. The value of δ_{ts} is defined based on the solution of the equation $F_m(\delta_{ts}) = m \cdot \delta_{ts}$ where m is some integer constant, $m > 1$. One of the solutions of $F_m(\delta_{ts}) = m \cdot \delta_{ts}$ is $\delta_{ts} = \frac{3m-2-\sqrt{5m^2-8m+4}}{2m-2}$ for $m > 1$. The value of δ_{ts} for policy ISM-DS, where $m > 0$, is $\delta_{ts} = B(m)$ and $B(m)$ is defined as follows:

$$B(m) = \begin{cases} 1 & \text{if } m = 1 \\ \frac{3m-2-\sqrt{5m^2-8m+4}}{2m-2} & \text{if } m > 1 \end{cases} \tag{5.12}$$

Note that the threshold density $B(m)$ can be determined based on the number of processors m. The two following inequalities in Eq. (5.13) and Eq. (5.14) hold for $B(m)$ and $B(m')$ where $1 \leq m' \leq m$:

$$B(m) \leq \frac{m}{2m-1} \leq \frac{m'}{2m'-1} \tag{5.13}$$

$$B(m) \leq B(m') \tag{5.14}$$

The proofs that Eq. (5.13) and Eq. (5.14) hold are given in Lemma A.1 and Lemma A.2 in the Appendix A (page 221, 222). Based on the threshold density $B(m)$, the priority assignment policy ISM-DS is given as follows:

ISM-DS Priority Assignment Policy: Given the number of processors m, the threshold density $\delta_{ts} = B(m)$ is calculated based on Eq. (5.12). The priorities to the tasks in set Γ are assigned as follows:

If $\delta_i > B(m)$, then task τ_i has the highest fixed priority (ties broken arbitrarily), otherwise, if $\delta_i \leq B(m)$, then task τ_i is given slack-monotonic priority.

Example 5.1. As an example of the way fixed priorities are assigned using the priority assignment policy `ISM-DS`, consider the following constrained-deadline task system to be scheduled on $m = 3$ processors based on global FP scheduling where the parameters of each task $\tau_i(C_i, D_i, T_i)$ are as follows:

$$\begin{aligned}\Gamma \stackrel{\text{def}}{=} \{&\tau_1 = (1, 2, 3) \quad \tau_2 = (2, 3, 5) \quad \tau_3 = (7, 100, 100)\\ &\tau_4 = (1, 25, 50) \quad \tau_5 = (2, 9, 10)\}\end{aligned}$$

The threshold density δ_{ts} is equal to $B(3) = 0.5$ for $m = 3$. The densities of the five tasks are $\delta_1 = 0.5$, $\delta_2 \approx 0.67$, $\delta_3 = 0.07$, $\delta_4 = 0.04$, and $\delta_5 \approx 0.23$. Since $\delta_2 > B(3)$, task τ_2 is assigned the highest fixed priority and each of the remaining tasks having density not greater than $B(3)$ is assigned the slack-monotonic priorities. The slack, i.e., $(D_i - C_i)$, of the remaining tasks τ_1, τ_3, τ_4 and τ_5 are respectively 1, 93, 24, and 7. Therefore, the final fixed priority ordering of all the tasks according to `ISM-DS` is given as (highest-priority task listed first): $\tau_2, \tau_1, \tau_5, \tau_4, \tau_3$ □

The global FP scheduler dispatches the tasks based on the priority assignment given by policy `ISM-DS`. Now the schedulability test in terms of *density bound* of global FP scheduling for the priority assignment policy `ISM-DS` is given in Theorem 5.2.

Theorem 5.2 (Density-Bound-Based Test)**.** *An constrained-deadline sporadic task set* Γ *is schedulable using global* FP *scheduling that assigns the priorities based on policy* `ISM-DS` *if the following condition, for* $m \geq 2$*, holds:*

$$\delta^n_{sum} \leq m \cdot min\{1/2, B(m)\}$$

where δ^n_{sum} *is the total density of the task set* Γ*.*

Proof. Given the task set Γ and the number of processors m, the two subsets Γ_L and Γ_H based on the threshold density $\delta_{ts} = B(m)$ are determined such that $\Gamma = \Gamma_L \cup \Gamma_H$. Remember that based on policy `ISM-DS` the tasks in set Γ_L and Γ_H are given the slack-monotonic and the highest fixed priorities, respectively. It will be shown that if the total density $\delta^n_{sum} \leq m \cdot min\{1/2, B(m)\}$, then the two general conditions **C1** and **C2** of Lemma 5.6 hold; which guarantee the schedulability of Γ using global FP scheduling.

(C1 holds) It is easy to see that $B(m) \geq min\{1/2, B(m)\}$. Then it follows that each task in Γ_H has density greater than $min\{1/2, B(m)\}$ since each task in Γ_H has density greater than $\delta_{ts} = B(m)$ for priority assignment policy `ISM-DS`. Since the total density (i.e., δ^n_{sum}) of task set Γ is not greater than $m \cdot min\{1/2, B(m)\}$ according to the premise, the number of tasks that are given the highest priority is less than m (**C1** holds).

(C2 holds) To show that **C2** of Lemma 5.6 holds, it will be shown that Γ_L is special on m' processors where $m' = (m - |\Gamma_H|)$. Let `DL` be the total density of all the tasks

in Γ_L. Also let δ_{maxL} and δ_{minL} be the maximum and minimum density of any task in set Γ_L, respectively. To show that Γ_L is special on m' processors, it will be shown that Property 1 and Property 2 (given in Definition 5.1, page 53) of special task set are satisfied. In other words, we have to show that the following two inequalities hold:

$$\textbf{Property 1} \quad \delta_{maxL} \leq \frac{m'}{2m'-1}$$
$$\textbf{Property 2} \quad \mathtt{DL} \;\leq\; \mathit{min}\{F_{m'}(\delta_{minL}), F_{m'}(\delta_{maxL})\}$$

(Property 1 holds for Γ_L) According to the priority assignment policy `ISM-DS`, no task in Γ_L has density greater than the threshold density $\delta_{ts} = B(m)$. So, we have $\delta_{maxL} \leq B(m)$. Moreover, from Eq. (5.13), we have $B(m) \leq \frac{m'}{2m'-1}$. Consequently, $\delta_{maxL} \leq \frac{m'}{2m'-1}$, and thus, Property 1 is satisfied for Γ_L.

(Property 2 holds for Γ_L) The total density of the tasks in Γ_H is greater than $(|\Gamma_H| \cdot min\{1/2, B(m)\})$ because each task in Γ_H has density greater than $\delta_{ts} = B(m)$ and $B(m) \geq min\{1/2, B(m)\}$. Since the total density of task set Γ is not greater than $m \cdot min\{1/2, B(m)\}$ according to the premise, the total density of the tasks in set Γ_L is at most $m' \cdot min\{1/2, B(m)\}$ where $m' = (m - |\Gamma_H|)$. Therefore, Eq. (5.15) holds.

$$\mathtt{DL} \leq m' \cdot min\{1/2, B(m)\} \tag{5.15}$$

Based on the threshold density $\delta_{ts} = B(m)$ of priority assignment policy `ISM-DS`, we have $\delta_{maxL} \leq B(m)$ since the density of any task in set Γ_L is not greater than $B(m)$. Moreover, from Eq. (5.14), we have $B(m) \leq B(m')$. Thus, $\delta_{maxL} \leq B(m')$.

It follows from Eq. (5.13) that $B(m') \leq \frac{m'}{2m'-1}$ (by replacing m by m' in the left-hand side inequality in Eq. (5.13)). Therefore, $\delta_{maxL} \leq B(m') \leq \frac{m'}{2m'-1}$. Because $0 \leq \delta_{minL} \leq \delta_{maxL}$, the inequality in Eq. (5.16) holds.

$$0 \leq \delta_{minL} \leq \delta_{maxL} \leq B(m') \leq \frac{m'}{2m'-1} \tag{5.16}$$

Based on Eq. (5.16) and from Eq. (5.5) of Lemma 5.3, the following inequality holds:

$$\mathit{min}\{F_{m'}(0)\,,\, F_{m'}(B(m'))\} \leq \mathit{min}\{F_{m'}(\delta_{minL})\,,\, F_{m'}(\delta_{maxL})\} \tag{5.17}$$

From the function definition given in Eq. (5.3), we have

$$F_{m'}(0) = \frac{m'(1-0)}{2-0} + 0 = m'/2 = m' \cdot 1/2 \tag{5.18}$$

It follows from Eq. (5.12) that $B(m') = 1$ when $m' = 1$. Thus, by setting $x = B(m')$ in Eq. (5.3) when $m' = 1$, we have $F_{m'}(B(m'))$=$F_1(1)$= $1 = m'$.
And for $m' > 1$, we have $F_{m'}(B(m')) = m' \cdot B(m')$ because one of the solutions

of function $F_{m'}(x) = m'x$ in terms of x is $x = B(m')$. Thus, for any $m' \geq 1$, the following inequality holds:

$$F_{m'}(B(m')) \geq m' \cdot min\{1, B(m')\} \tag{5.19}$$

It follows from Eq. (5.18) and Eq. (5.19) that

$$min\{F_{m'}(0), F_{m'}(B(m'))\} \geq m' \cdot min\{1/2, B(m')\} \tag{5.20}$$

Then it follows from Eq. (5.20) and the fact that $B(m) \leq B(m')$ in Eq. (5.14) that

$$m' \cdot min\{1/2, B(m)\} \leq min\{F_{m'}(0), F_{m'}(B(m'))\} \tag{5.21}$$

Thus, it now follows from Eq. (5.15) and Eq. (5.21) that

$$\texttt{DL} \leq min\{F_{m'}(0), F_{m'}(B(m'))\ \} \tag{5.22}$$

Finally, from Eq. (5.17) and Eq. (5.22), we have

$$\texttt{DL} \leq min\{F_{m'}(\delta_{minL}), F_{m'}(\delta_{maxL})\ \} \tag{5.23}$$

Therefore, Property 2 is satisfied for task set Γ_L (i.e., **C2** holds). Consequently, if $\delta^n_{sum} \leq m \cdot min\{1/2, B(m)\}$, then the task set Γ is schedulable using global FP scheduling where priorities are assigned based on `ISM-DS` policy. □

The density bound $m \cdot min\{1/2, B(m)\}$ of global FP scheduling of constrained-deadline sporadic tasks, for any finite $m \geq 2$, using policy `ISM-DS` is greater than or equal to the state-of-the-art density bound $\frac{m+1}{3}$ for `DM-DS[`$\frac{1}{3}$`]` scheduling. Figure 5.1 illustrates the density bounds of `DM-DS[`$\frac{1}{3}$`]` and `ISM-DS` for $m = 2, \ldots 16$. The x-axis in Figure 5.1 represents the number of processors and the y-axis represents the density bound normalized by number of processors.

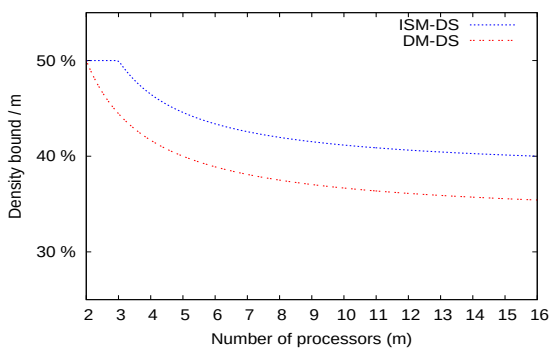

Figure 5.1: *Density bounds of* `DM-DS[`$\frac{1}{3}$`]` *and* `ISM-DS`.

The total density of the task set in Example 5.1 (page 59) is ≈ 1.499. The density bound $m \cdot min\{1/2, B(m)\}$ using ISM-DS policy for $m = 3$ is 1.5. Therefore, the task set in Example 5.1 is global FP schedulable using ISM-DS priority assignment policy. The DM-DS$[\frac{1}{3}]$ scheduling can not guarantee the schedulability of the task set in Example 5.1 since the density bound $\frac{m+1}{3}$ for DM-DS$[\frac{1}{3}]$ is ≈ 1.33.

5.5 Policy ISM-DS$[\xi]$: Searching the Threshold

The threshold density used for priority assignment policy ISM-DS depends only on the number of processors and does not use any information (e.g., density) of individual task of the given task set. Using the density information of individual task in addition to the information about the number of processors, a better threshold density can be searched from the set of densities of the tasks for assigning the priorities based on slack-monotonic HPA policy. This new priority assignment policy is called ISM-DS$[\xi]$ where the threshold density ξ is searched among the densities of the tasks in a task set. It will be shown that the schedulability test of global FP scheduling using policy ISM-DS$[\xi]$ dominates and empirically performs much better than that of using ISM-DS.

Remember that based on the Observation 5.1 (page 58), the HPA policy can guarantee the schedulability of a task set using global FP scheduling if k tasks are given the highest fixed priorities and the remaining $(n - k)$ tasks are global FP schedulable on $(m-k)$ processors using some other fixed priority assignment, for some k, $0 \leq k < m$. The proposed priority assignment policy ISM-DS$[\xi]$ is based on a similar technique used for priority assignment in priority-driven scheduling, called EDF$^{(k)}$, proposed by Goossens et al. [GFB03] for implicit-deadline tasks. In EDF$^{(k)}$ scheduling, the jobs of the k highest utilization tasks are given the highest priority and the jobs of the remaining $(n - k)$ lowest utilization tasks are given the EDF priorities for some appropriate selection of k, $0 \leq k < m$. Inspired by the priority assignment scheme for EDF$^{(k)}$ scheduling, the slack-monotonic HPA policy ISM-DS$[\xi]$ for constrained-deadline sporadic tasks is defined as follows:

1. *Each of the k highest density tasks is given the highest fixed priority, and*
2. *the remaining $(n-k)$ lowest density tasks are given the slack-monotonic priorities for some k such that $0 \leq k < m$.*

The challenge for ISM-DS$[\xi]$ priority assignment policy is to find an appropriate k, where $0 \leq k < m$, to guarantee the schedulability. Note that after the value of k is known, the density of the $(k + 1)^{th}$ highest density task is the threshold density ξ for priority assignment policy ISM-DS$[\xi]$. For example, if $k = 0$, then the largest density of any tasks in the task set is used as the threshold density (i.e., all tasks are given SM priority). If $k = 1$, then the second largest density of the tasks in a task set is used as the threshold density (i.e., only the largest density task is assigned the highest-fixed

priority and the remaining tasks are given the slack-monotonic priorities). The challenge is how to find such k, if exists, that would guarantee the schedulability of the entire task set. The pseudocode to search such k, where $k < m$, for the priority assignment policy ISM-DS[ξ] is presented in algorithm Find(ξ) in Figure 5.2. Algorithm Find(ξ) determines if there is some k, $0 \leq k < m$, such that entire task set is schedulable using the priority assignment policy ISM-DS[ξ]. The search for the k in algorithm Find(ξ) is guided by the following schedulability condition given in Theorem 5.3.

Theorem 5.3. *A constrained-deadline sporadic task set* Γ *is schedulable using global* FP *scheduling algorithm according to the priority assignment policy* ISM-DS[ξ] *if the set of* $(n-k)$ *lowest density tasks of task set* Γ *is special on* $(m-k)$ *processors for some* k*, where* $0 \leq k < m$*.*

Proof. Using policy ISM-DS[ξ], the $(k+1)^{th}$ highest density task in task set Γ is used as the threshold density δ_{ts} for some k, $0 \leq k < m$. The threshold density δ_{ts} decides the tasks in set Γ_L and Γ_H that are respectively given the slack-monotonic and the highest fixed priorities such that $\Gamma = \Gamma_L \cup \Gamma_H$.

Note that, using policy ISM-DS[ξ], the number of tasks having the highest fixed priority is $|\Gamma_H| = k$ for some k where $0 \leq k < m$. Consequently, condition **C1** of Lemma 5.6 is satisfied for policy ISM-DS[ξ]. According to Lemma 5.6, the value of k has to be chosen such that the condition **C2** of Lemma 5.6 holds to guarantee the schedulability of task set Γ using global FP scheduling. In other words, task set Γ can be guaranteed to be schedulable using global FP scheduling according to policy ISM-DS[ξ] whenever Γ_L is special on $(m-k)$ processors, where Γ_L contains all the $(n-k)$ lowest density tasks from set Γ. □

Deriving a k, if one exists, that satisfies Theorem 5.3 is straightforward. One such example algorithm (called Find(ξ)) that searches (if exists) the value of k is presented in Figure 5.2. The algorithm Find(ξ) returns **True** if it can find some k such that the set of $(n-k)$ lowest density tasks from set Γ is special on $(m-k)$ processors such that $0 \leq k < m$, otherwise, it returns **False**.

In line 1–2, algorithm Find(ξ) in Figure 5.2 initializes local variables Γ_L and Γ_H as $\Gamma_L = \Gamma$ and $\Gamma_H = \emptyset$ to consider first whether all the tasks in Γ are special on m processors (checked during the first iteration of the **For** loop in line 3–12).

The **For** loop in line 3–12 iterates at most m times for the iterative variable k that iterates form 0 to $(m-1)$. In each iteration of the **For** loop, it is checked that whether the $(n-k)$ lowest density tasks in set Γ_L are special on $(m-k)$ processors. Note that in order to determine whether Γ_L is special on $(m-k)$ processors, both Property 1 and Property 2 (Definition 5.1, page 53) of special task system have to be satisfied. If the task set Γ_L is special on $(m-k)$ processors (condition at line 4 is true), then slack-monotonic priorities are assigned to the tasks in Γ_L (line 5), each of the tasks in Γ_H is assigned the highest fixed priority (line 6) and the algorithm returns **True** (line 7).

During a particular iteration of the **For** loop, if the task set Γ_L is not special on $(m-k)$ processors (condition at line 4 is false), then the highest density task, say $\tau_{ts} \in \Gamma_L$, is extracted from Γ_L (line 9) and is included in set Γ_H (line 10). Note that at

Algorithm `Find(`ξ`)`

1. $\Gamma_H = \emptyset$
2. $\Gamma_L = \Gamma$
3. **For** $k = 0$ to $(m-1)$
4. **If** Γ_L is special on $(m-k)$ processors **Then**
5. **Print** "All tasks in Γ_L are assigned slack-monotonic priority"
6. **Print** "All tasks in Γ_H are assigned the highest fixed priority"
7. **Return True**
8. **End If**
9. Find τ_{ts} such that δ_{ts} is the largest density in set Γ_L
10. $\Gamma_H = \Gamma_H \cup \{\tau_{ts}\}$
11. $\Gamma_L = \Gamma - \Gamma_H$
12. **End For**
13. **Print** "Priority Assignment Fails"
14. **Return False**

Figure 5.2: *Slack-monotonic* HPA *by searching the threshold*

the beginning of the k^{th} iteration of the **For** loop, the largest density of the tasks in Γ_L is the $(k+1)^{th}$ largest density of the tasks in the entire task set Γ. At the beginning of each iteration of the **For** loop, total k largest density tasks are in set Γ_H and the remaining $(n-k)$ lowest density tasks are in set Γ_L. If the task set Γ_L is not special on $(m-k)$ processors for any k, such that $0 \leq k < m$, then policy `ISM-DS`$[\xi]$ fails to assign the fixed priorities to the tasks in Γ (line 13) and the algorithm returns **False** (line 14). By sorting the tasks in set Γ in order of increasing densities of the tasks, it is not difficult to see that algorithm `Find(`ξ`)`can be implemented using at most $O(n \cdot \log n)$ operations.

The schedulability test in Theorem 5.3 for global FP scheduling using priority assignment policy `ISM-DS`$[\xi]$ dominates that of the density-bound test in Theorem 5.2. Now it will be shown that any task set deemed schedulable based on Theorem 5.2 is also deemed schedulable using Theorem 5.3, and not conversely.

Assume a contradiction where a task set Γ is not guaranteed schedulable based on Theorem 5.3 for priority assignment policy `ISM-DS`$[\xi]$ but schedulable using Theorem 5.2 for `ISM-DS` priority assignment policy. If Γ is not guaranteed to be schedulable under `ISM-DS`$[\xi]$ based on schedulability test in Theorem 5.3, then there exist no k such that the set of $(n-k)$ lowest density tasks is special on $(m-k)$ processors for any $k < m$ (according to the contrapositive of Theorem 5.3).

When Γ is schedulable under `ISM-DS` based on Theorem 5.2, the proof of the schedulability condition in Theorem 5.2 guarantees that there exists a task set Γ_L that is special on $(m - |\Gamma_H|)$ processors and $|\Gamma_H| < m$. So, there exists some k such that the set of $(n-k)$ lowest density tasks is special on $(m-k)$ processors for some $k < m$ (contradiction!). Therefore, any task set schedulable using `ISM-DS` based on Theorem 5.2 is also schedulable using `ISM-DS`$[\xi]$ based on Theorem 5.3.

It will be shown using the following Example 5.2 that the converse is not true; that

is, there is a task set that is global FP schedulable based on the schedulability test in Theorem 5.3 for ISM-DS[ξ] policy but is not guaranteed to be schedulable based on the density-bound test in Theorem 5.2 for ISM-DS priority assignment policy.

Example 5.2. Consider $n = 11$ tasks in set $\Gamma = \{\tau_1, \ldots \tau_{11}\}$ such that $\delta_1 = \ldots = \delta_{10} = 0.40$ and $\delta_{11} = 0.15$. Thus, the total density of task set Γ is $\delta^n_{sum} = 4.15$. The task set Γ is to be scheduled using global FP scheduling on $m = 10$ processors.

Notice that Property 1 of special task system is satisfied for task set Γ because $\delta^n_{max} = 0.4 < m/(2m-1)$ for $m = 10$. Since $m = 10$, δ^n_{max}=0.40 and δ^n_{min}=0.15, we have $F_m(\delta^n_{min}) \approx 4.745$ and $F_m(\delta^n_{max})$ = 4.150. Consequently, it is true that $min\{F_m(\delta^n_{min}), F_m(\delta^n_{max})\} = 4.150$ and $\delta^n_{sum} \leq min\{F_m(\delta^n_{min}), F_m(\delta^n_{max})\}$ holds. So, the entire task set Γ is special on $m = 10$ processors and global FP schedulable based on Theorem 5.3 for ISM-DS[ξ] priority assignment policy.

However, the schedulability test in Theorem 5.2 is not satisfied for Γ (i.e., density bound $m \cdot min\{1/2, B(m)\} = 4.116 < \delta^n_{sum}$). Consequently, the schedulability of Γ using ISM-DS policy can not be guaranteed. So, the schedulability test for policy ISM-DS[ξ] in Theorem 5.3 dominates that of in Theorem 5.2 for ISM-DS. □

5.6 Empirical Investigation

In this section, empirical investigation into the two proposed schedulability tests for priority assignment policies ISM-DS and ISM-DS[ξ] is presented. In order to measure the improvement of these proposed tests over the state-of-the-art DM-DS[$\frac{1}{3}$] test, simulation using randomly generated task sets is conducted. The well-known metric, called *acceptance ratio*, is used to evaluate the effectiveness (in terms of determining schedulability of randomly generated task sets) of the three priority assignment policies and schedulability tests given in Table 5.1.

Priority Assignment Policy	Schedulability Test Used
DM-DS[$\frac{1}{3}$]	The density bound $\frac{m+1}{3}$ (proved in [BCL05]) is used as the schedulability test.
ISM-DS	The density bound $m \cdot min\{1/2, B(m)\}$ proved in Theorem 5.2 is used as the schedulability test.
ISM-DS[ξ]	Algorithm Find(ξ)in Figure 5.2 is used as the schedulability test.

Table 5.1: *Different priority assignment policies and the associated schedulability tests.*

The acceptance ratio of a schedulability test is the percentage of the randomly generated task sets that are deemed schedulable using that schedulability test at a particular utilization level. All the randomly generated task sets generated at a particular utilization level have the same total utilization. The acceptance ratios of the three priority assignment policies and schedulability tests in Table 5.1 are presented in this section. Before presenting the experimental results, the task set generation algorithm is presented next.

5.6.1 Task Sets Generation Algorithm

The `UUnifast` algorithm (given in Figure 5.3), which was originally proposed by Bini and Buttazzo [BB05] to generate utilizations of a task set to study uniprocessor scheduling, is adapted by Davis and Burns in [DB09] to generate utilizations of a task set to study multiprocessor scheduling.

Algorithm `UUnifast(n, U)`

```
1. SumU= U
2. For (i=0 to n-1)
3.   nextSumU = SumU * pow(rand(),1/(n-i));
4.   U[i]=SumU-nextSumU;
5.   SumU=nextSumU;
6. End For
7. U[n]=SumU;
```

Figure 5.3: *The `UUnifast` algorithm [BB05]. The function `pow(x,y)` returns x^y and `rand()` returns a random number in the range [0,1].*

Based on the `UUnifast` algorithm, Davis and Burns proposed the following three steps (called the `UUnifast-Discard` algorithm) to generate a task set with cardinality n and total utilization U to study scheduling on multiprocessors:

- **Step 1:** The `UUnifast` algorithm with parameters n and U is used to generate task utilization values in the range $[0, U]$.
- **Step 2:** If the utilization of a task is greater than 1, then the utilization values produced so far are discarded. If the total number of such discarded partial task sets exceeds some limit, say DISCARD_{lim}, then the algorithm exits by reporting *failure*, otherwise, Step 1 is re-executed.
- **Step 3:** If the utilization of no task is greater than 1, then a set of n valid utilization values are generated and the algorithm exits by reporting *success*.

The derivation of this task set generation algorithm to study multiprocessor scheduling is motived by the following reason as pointed out by Davis and Burns in [DB11b]:

> "*A task set generation algorithm should be unbiased ... and ... should allow task sets to be generated that comply with a specified parameter setting. That way the dependency of priority assignment policy / schedulability test effectiveness on each task set parameter can be examined by varying that parameter, while holding all other parameters constant, avoiding any confounding effects.*"

It is proved in [DB09, DB11b] that the `UUnifast-Discard` algorithm generates an unbiased (i.e., uniformly distributed [BB05]) task set with cardinality n where each task's utilization is in the range $[0, min\{U, 1\}]$ and total utilization of the task set is U.

In this thesis, the UUnifast-Discard algorithm is used to generate n utilization values of a task set using $\text{DISCARD}_{lim} = 1000$. Once a set of n utilizations $\{u_1, u_2, \ldots u_n\}$ of a task set is generated, the other parameters of each task τ_i in the task set are generated as follows:

- The minimum inter-arrival time T_i of each task τ_i is generated from the uniform random distribution within the range $[10ms, 1000ms]$.
- The WCET of task τ_i is set to $C_i = u_i \cdot T_i$.
- The relative deadline D_i of task τ_i is generated from the uniform random distribution within the range $[C_i, T_i]$.

Each of the experiments is characterized by a pair (m, n) where m is the number of processors and n is the cardinality of task set. For each experiment (m, n), task sets are generated at 40 different utilization levels: $\{0.025m, 0.5m, \ldots 0.975m, m\}$. A total of 1000 task sets at each of the 40 utilization levels using the UUnifast-Discard algorithm with parameters n and U are generated. Each of the 1000 task sets generated at a particular utilization level, say U, has cardinality n and total utilization equal to U. The schedulability of each of the 1000 task sets generated at each utilization level are determined based on the schedulability test for each of the three priority assignment policies in Table 5.1 and the acceptance ratio for each test is computed.

5.6.2 Result Analysis

A series of experiments are conducted using randomly generated task sets for different pairs of (m, n) where $m \in \{2, 4, 8, 16\}$ and $n \in \{2.5m, 5m, 10m\}$. The acceptance ratios of three experiments with parameters $(m = 4, n = 10)$, $(m = 4, n = 20)$, and $(m = 4, n = 40)$ are given in Figure 5.4–5.6. And, the acceptance ratios of three experiments $(m = 8, n = 20)$, $(m = 8, n = 40)$, and $(m = 8, n = 80)$ are given in Figure 5.7–5.9. The important trends and observations based on these experiments are presented in this section; and the results of other experiments follow a similar trend.

Observation 1: The schedulability test of the ISM-DS$[\xi]$ priority assignment policy significantly outperforms that of both DM-DS$[\frac{1}{3}]$ and ISM-DS priority assignment policies. In the experiment $(m = 4, n = 20)$ in Figure 5.5, the acceptance ratio at utilization level $0.275m$ is approximately 0% using the schedulability tests for both DM-DS$[\frac{1}{3}]$ and ISM-DS priority assignment policies while the acceptance ratio of the schedulebality test for ISM-DS$[\xi]$ poiority assignment policy is more than 70%. This is due to the improved priority assignment policy ISM-DS$[\xi]$ that searches the threshold density by taking into consideration of the densities of the tasks in addition to the number of processors.

Observation 2: The acceptance ratios for all the tests decreases as the number of tasks in a task set increases where m is constant. This is because the total density of the task set at each utilization level generally increases as the number of tasks in a task set

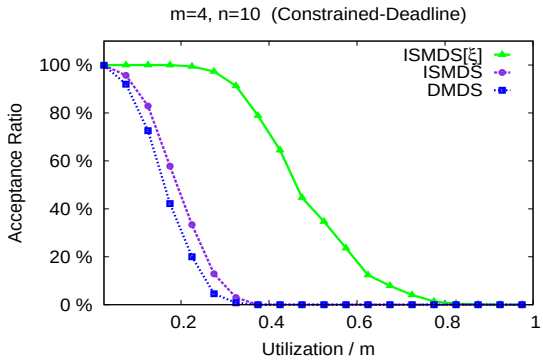

Figure 5.4: *Acceptance ratios for experiments with* $(m = 4, n = 10)$.

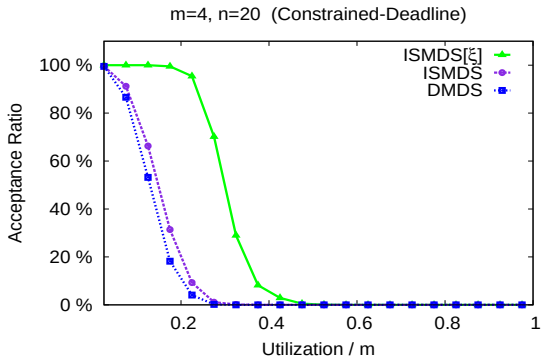

Figure 5.5: *Acceptance ratios for experiments with* $(m = 4, n = 20)$.

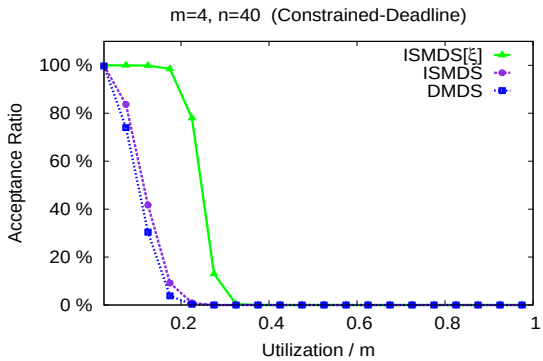

Figure 5.6: *Acceptance ratios for experiments with* $(m = 4, n = 40)$.

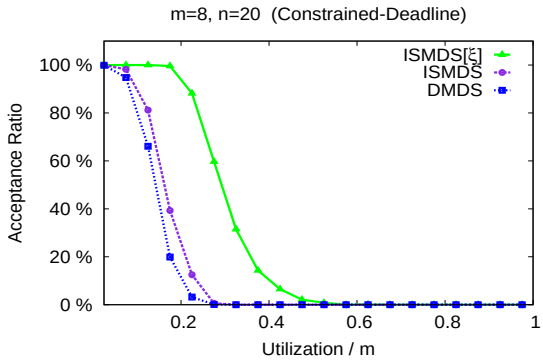

Figure 5.7: *Acceptance ratios for experiments with* $(m = 8, n = 20)$.

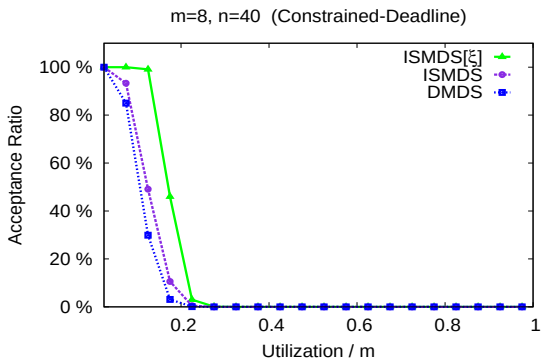

Figure 5.8: *Acceptance ratios for experiments with* $(m = 8, n = 40)$.

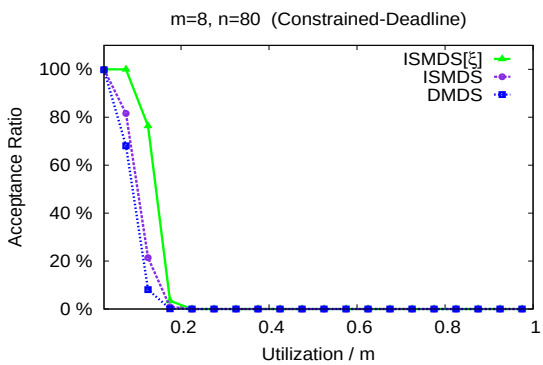

Figure 5.9: *Acceptance ratios for experiments with* $(m = 8, n = 80)$.

increases. This conclusion is made based on another set of experiments that verifies that the fact the total density of task set at each utilization level generally increases due to the increase in cardinality of the task set. The *normalized average density* of 1000 task sets at each utilization level is computed for experiments $(m = 8, n)$ for five different values of $n = 10, 20, 40, 60, 80$. The normalized average density is calculated as follows: the total density of 1000 task sets at each utilization level is first divided by 1000 to compute the average density which is then divided by m.

Figure 5.10 plots the normalized average density on the y-axis and the normalized utilization level on the x-axis for experiments with $m = 8$ and $n = 10, 20, 40, 60, 80$. Similar result is also shown in Figure 5.11 for experiments with $m = 4$ and $n = 8, 10, 20, 30, 40$.

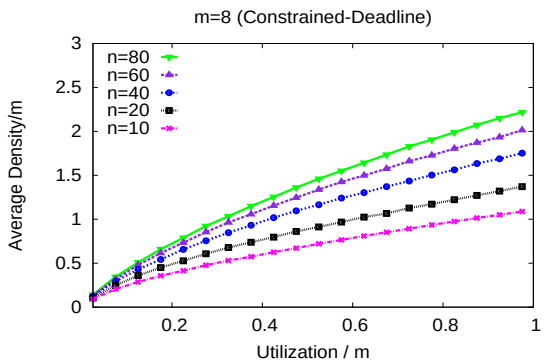

Figure 5.10: *Increase in normalized average density with the increase in task set cardinality for experiments with* $m = 8$ *and* $n = 10, 20, 40, 60, 80$.

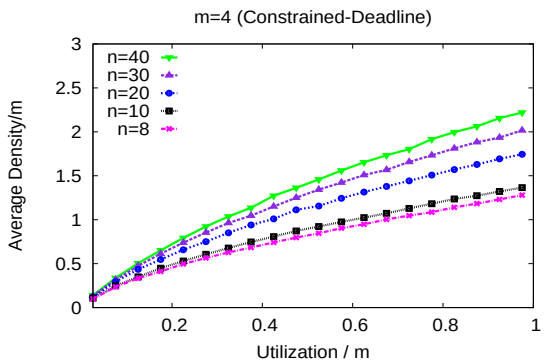

Figure 5.11: *Increase in normalized average density with the increase in task set cardinality for experiments with* $m = 4$ *and* $n = 8, 10, 20, 30, 40$.

It is evident that the average density of a task set increases as the number of tasks in a task set increases for each utilization level and fixed number of processors. Since the three schedulability tests in Table 5.1 highly depend on the total density of a task set and because a constrained-deadline task set with relatively higher density is more difficult to schedule, the acceptance ratio decreases as the number of tasks in a task set increases for a given number of processors.

Observation 3: The acceptance ratios of the two schedulability tests for DM-DS[$\frac{1}{3}$] and ISM-DS priority assignment policies increase slightly due to the increase in number of processors while the task set cardinality does not change (compare the acceptance ratios for experiments $(m = 4, n = 20)$ and $(m = 8, n = 20)$ in Figure 5.5 and Figure 5.7, respectively). This is because the normalized average density of a task set decreases as the number of processors increases while keeping the task set cardinality constant. Figure 5.12 plots the normalized average density against the normalized utilization level for experiments with $n = 40$ and $m = 2, 4, 8, 16$. It is evident that for a given cardinality of the task set, the normalized average density of a task set decreases at each utilization level with the increase in number of processors. Consequently, the acceptance ratio of the density-based tests for policies DM-DS[$\frac{1}{3}$] and ISM-DS increases with the increase in number of processors for some fixed cardinality of the task sets.

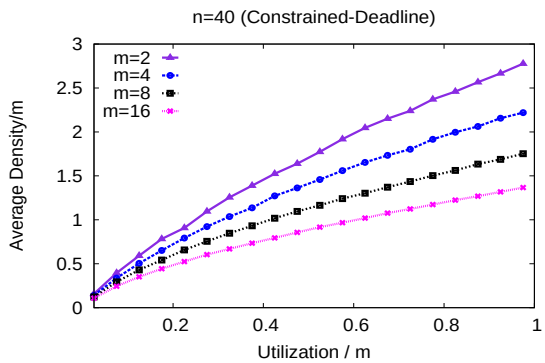

Figure 5.12: *Decrease in normalized average density with the increase in number of processors for experiments with task set cardinality $n = 40$ and $m = 2, 4, 8, 16$.*

Observation 4: The acceptance ratios of schedulability test for ISM-DS[ξ] priority assignment policy decreases noticeably with the increase in number of processors while keeping the task set cardinality constant (compare the acceptance ratios of ISM-DS[ξ] priority assignment policy for experiments $(m = 4, n = 40)$ and $(m = 8, n = 40)$ in Figure 5.6 and Figure 5.8, respectively). If the number of processors increases from one experiment to another, the total utilization of the task sets generated at each normalized utilization level also increases. Task set with relatively larger total utilization also has relatively larger total density. Consequently, the number of tasks with relatively larger

individual density in a task set increases as the total density of the task set increases while the task set cardinality remains constant. If the individual density of each task in a task set is relatively larger, then the algorithm `Find(`ξ`)` in Figure 5.2 often fail to find any k, $0 \leq k < m$, such that the set of $(n-k)$ lowest density tasks is special on $(m-k)$ processors. In other words, task set having higher number of high density tasks suffers from Dhall's effect and can not be guaranteed schedulable using the schedulability test for `ISM-DS`$[\xi]$ priority assignment policy.

5.7 Implicit-Deadline Tasks: Utilization Bound

The priority assignment policy `ISM-DS` is also applicable to implicit-deadline task sets. Note that the density and utilization of implicit-deadline task systems are the same. The schedulability test for implicit-deadline tasks is called the utilization-bound test which is given in Theorem 5.4 (proof is obvious by considering $D_i = T_i$ in Theorem 5.3).

Theorem 5.4. *An implicit-deadline sporadic task system* Γ *is schedulable using global* FP *scheduling that assigns the priorities based on policy* `ISM-DS` *if the following condition, for* $m \geq 2$*, holds:*

$$U^n \leq m \cdot min\{1/2, B(m)\}$$

where U^n *is the total utilization of the task set* Γ.

Example 5.3. As an example of the way fixed priorities are assigned using the priority assignment policy `ISM-DS`, consider the following implicit-deadline task system to be scheduled on $m = 3$ processors based on global FP scheduling where the parameters of each task $\tau_i(C_i, T_i)$ are as follows:

$$\begin{aligned}\Gamma \stackrel{\text{def}}{=} \{&\tau_1 = (1,2) \quad \tau_2 = (2,3) \quad \tau_3 = (7,100)\\ &\tau_4 = (1,25) \quad \tau_5 = (2,9)\}\end{aligned}$$

The threshold density or utilization δ_{ts} is equal to $B(3) = 0.5$. The utilizations of the five tasks are $u_1 = 0.5$, $u_2 \approx 0.67$, $u_3 = 0.07$, $u_4 = 0.04$, and $u_5 \approx 0.23$. Since $u_2 > B(3)$, task τ_2 is assigned the highest fixed priority. The slack of tasks τ_1, τ_3, τ_4 and τ_5 are respectively 1, 93, 24, and 7. Therefore, the final fixed priority ordering of all the tasks are as follows (highest-priority task listed first): $\tau_2, \tau_1, \tau_5, \tau_4, \tau_3$. □

The utilization bound $m \cdot min\{1/2, B(m)\}$ of global FP scheduling, for any finite $m \geq 2$, using policy `ISM-DS` is higher than the state-of-the-art utilization bounds $\frac{m+1}{3}$ and $\frac{2m}{3+\sqrt{5}}$ of `RM-US` $[\frac{1}{3}]$ and `SM-US`$[\frac{2}{3+\sqrt{5}}]$ scheduling, respectively. Figure 5.13 illustrates the utilization bounds of `RM-US`$[\frac{1}{3}]$, `SM-US`$[\frac{2}{3+\sqrt{5}}]$ and `ISM-DS` for $m = 2, \ldots 16$. The x-axis in Figure 5.13 represents the number of processors and the y-axis represents the utilization bound normalized by number of processors for different

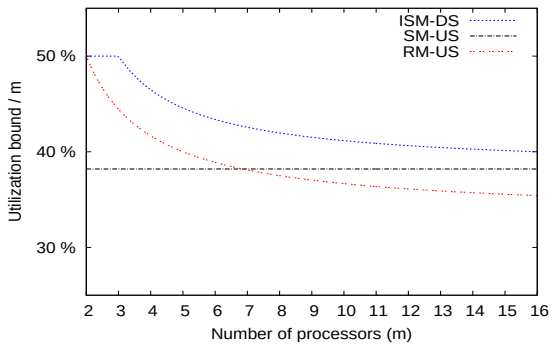

Figure 5.13: *Utilization bounds of* `RM-US`$[\frac{1}{3}]$, `SM-US`$[\frac{2}{3+\sqrt{5}}]$ *and* `ISM-DS`.

priority assignments. Notice that the proposed bound is same as for `RM-US` $[\frac{1}{3}]$ when $m = 2$ and the same as for `SM-US`$[\frac{2}{3+\sqrt{5}}]$ when $m = \infty$.

The total utilization of the task set in Example 5.3 is ≈ 1.499. The utilization bound $m \cdot min\{1/2, B(m)\}$ using `ISM-DS` policy for $m = 3$ is 1.5. Therefore, the task set in Example 5.3 is global FP schedulable using `ISM-DS` priority assignment policy. Neither `RM-US` $[\frac{1}{3}]$ nor `SM-US`$[\frac{2}{3+\sqrt{5}}]$ can guarantee the schedulability of this task set since the utilization bound for these policies are ≈ 1.33 and 1.14, respectively.

5.7.1 Independent and Scale Invariant Priority Assignment

In this subsection, the best achievable utilization bound of global FP scheduling of implicit-deadline task sets, where no task's utilization is in the range $(1-\frac{1}{\sqrt{2}}, \sqrt{2}-1] \approx (0.293, 0.414]$, is proposed for the class of fixed-priority assignment policies that are independent and scale invariant. A priority assignment scheme is *independent* [AJ03] if the priority of a task τ_i depends only on its own parameters, i.e., the priorities of tasks are assigned according to the function $prio_i = f(T_i, C_i)$. A priority assignment scheme is *scale-invariant* [AJ03] if the relative priority order of the tasks does not change when the T_i and C_i of all the tasks are multiplied by the same positive constant. In other words, $f(T_i, C_i)$ is scale invariant if and only if the following holds for all $A > 0$:

$$f(T_i, C_i) < f(T_j, C_j) \Leftrightarrow f(A \cdot T_i, A \cdot C_i) < f(A \cdot T_j, A \cdot C_j)$$

Andersson and Jonsson showed in [AJ03] that the utilization bound for global FP scheduling of implicit-deadline task set using an independent and scale-invariant priority assignment scheme can not be greater than $(\sqrt{2}-1)m \approx 0.414m$. The problem of determining such an independent and scale-invariant priority assignment scheme with a utilization bound of $(\sqrt{2}-1)m$ for global FP scheduling is still open.

In the *First International Real-Time Scheduling Open Problems Seminar* held in conjunction with the 22nd Euromicro Conference on Real-Time Systems (ECRTS) in Belgium, 2010, Andersson presented a conjecture regarding this open problem [And10]:

the utilization bound of slack-monotonic HPA policy using $(\sqrt{2}-1)$ as the threshold utilization is $(\sqrt{2}-1)m$ for implicit-deadline task systems (called, the SM-US$[\sqrt{2}-1]$ priority assignment scheme). The SM-US$[\sqrt{2}-1]$ priority assignment policy assigns the highest fixed priority to each task having utilization greater than $(\sqrt{2}-1)$ and each of the remaining tasks is assigned lower, slack-monotonic priorities. While the problem of proving this conjecture is still open for arbitrary task sets, this problem is closed in this thesis for task sets in which no task has utilization within the range $(1-\frac{1}{\sqrt{2}}, \sqrt{2}-1]$.

Theorem 5.5. *An implicit-deadline sporadic task set Γ is schedulable using global* FP *scheduling under SM-US$[\sqrt{2}-1]$ priority assignment policy, if the following condition, for $m \geq 2$, holds:*

$$U^n \leq m \cdot (\sqrt{2}-1)$$

where $u_i \leq (1-\frac{1}{\sqrt{2}})$ or $u_i > (\sqrt{2}-1)$ for each $\tau_i \in \Gamma$.

Proof. The proof is given in Appendix A (page 223). □

If the utilization-bound test in Theorem 5.4 can not guarantee the global FP schedulability of an implicit-deadline task set where no task's utilization is in the range $(1-\frac{1}{\sqrt{2}}, \sqrt{2}-1] \approx (0.293, 0.414]$, then Theorem 5.5 can be used to test the schedulability of the task set. For such task sets, where no task's utilization is in the range $(1-\frac{1}{\sqrt{2}}, \sqrt{2}-1]$, we have at our disposal a priority-assignment scheme that attains the best utilization bound possible for the class of independent and scale invariant fixed-priority assignment schemes for global FP scheduling.

The utilization bound of ISM-DS priority assignment policy for arbitrary task sets is greater than $m\cdot(\sqrt{2}-1)$ whenever $m \leq 9$. Therefore, the utilization bound of $m\cdot(\sqrt{2}-1)$ for SM-US$[\sqrt{2}-1]$ priority assignment policy is useful to test the schedulability only for task set where no task's utilization is in the range $(1-\frac{1}{\sqrt{2}}, \sqrt{2}-1]$ and $m \geq 10$. No task sets with total utilization $m \cdot (\sqrt{2}-1)$ for $m \geq 10$ passes the utilization bound test for the ISM-DS priority assignment policy. However, such task set with total utilization $m\cdot(\sqrt{2}-1)$ passes the utilization bound test of the SM-US$[\sqrt{2}-1]$ priority assignment policy if no task's utilization is in the range $(1-\frac{1}{\sqrt{2}}, \sqrt{2}-1]$.

	$m=16$	$m=32$
$n=3m$	2.8%	0%
$n=5m$	13.1%	1.9%
$n=8m$	67.6%	43.4%
$n=10m$	89.5%	79.4%
$n=15m$	99.6%	98.4%

Table 5.2: *Acceptance ratios, based on the schedulability test in Theorem 5.5, of the 1000 randomly generated task sets each with total utilization $m(\sqrt{2}-1)$.*

The acceptance ratios using the schedulability test in Theorem 5.5 of 1000 randomly generated task sets, each with total utilization $m \cdot (\sqrt{2}-1)$ for $m = 16, 32$ and $n =$

$3m, 5m, 8m, 10m, 15m$, are computed and presented in Table 5.2. As the number of tasks in a task set, each having a total utilization $m(\sqrt{2}-1)$ increases, the possibility of having a task with utilization greater than $1-\frac{1}{\sqrt{2}}$ decreases and the acceptance ratio increases.

5.8 Uniprocessor Slack-Monotonic Scheduling

It has been proved by Andersson in [And08b] that the utilization bound for *uniprocessor* slack-monotonic scheduling of implicit-deadline task set is 50%. The schedulability analysis of "special" task system on multiprocessors proposed in this thesis (Section 5.4.2, page 53) enables the derivation of a higher utilization bound for uniprocessor slack-monotonic scheduling compared to that of the state-of-the-art result in [And08b].

First, it will be shown below that the density bound for slack-monotonic scheduling of constrained-deadline tasks on uniprocessor is $F_1(\delta^n_{min})$. Then, the corresponding utilization bound $F_1(u^n_{min})$ for implicit-deadline tasks is shown to dominate the state-of-the-art bound of 50% for slack-monotonic uniprocessor scheduling.

Consider a task system Γ that is special on uniprocessor (i.e. $m = 1$). According to Property 1 of special task system Γ (Definition 5.1, page 53), we have $\delta^n_{max} \leq 1$ because $m/(2m-1) = 1$ for $m = 1$. Therefore, special task system Γ is in fact an arbitrary task system for uniprocessor slack-monotonic scheduling whenever $m = 1$ since there is no restriction on the maximum density of individual task. Note that we have $0 < \delta^n_{min} \leq \delta^n_{max} \leq 1$ where δ^n_{min} and δ^n_{max} are the minimum and maximum density of any task in Γ, respectively.

For $m = 1$, the function $F_1(x)$ is increasing within $[0, 1]$ since $F_1'(x) = 1 - \frac{1}{(2-x)^2} > 0$ within $(0, 1)$. Consequently, $min\{F_1(\delta^n_{min}),F_1(\delta^n_{max})\} = F_1(\delta^n_{min})$ since $\delta^n_{min} \leq \delta^n_{max}$. It is obvious from Property 2 of special task system Γ that for $m = 1$ that

$$\delta^n_{sum} \leq min\{F_1(\delta^n_{min}),F_1(\delta^n_{max})\} = F_1(\delta^n_{min}) \tag{5.24}$$

Using Theorem 5.1, the special task set Γ is schedulable using $\mathsf{GS}_{\mathsf{SM}}$ (i.e., uniprocessor slack-monotonic scheduling when $m = 1$). Therefore, the density bound for uniprocessor slack-monotonic scheduling of constrained-deadline task is $F_1(\delta^n_{min})$. Evidently, the utilization bound for uniprocessor slack-monotonic scheduling of implicit-deadline task sets is $F_1(u^n_{min})$.

The current state-of-the-art utilization bound for SM uniprocessor scheduling of implicit-deadline tasks is 50% which is proposed in [And08b]. It will now be shown that $F_1(u^n_{min}) > 50\%$. Since the function $F_1(x)$ is increasing within $[0, 1]$, we have $F_1(u^n_{min}) > F_1(0)$ since $u^n_{min} > 0$. Note that $F_1(0) = \frac{1(1-0)}{2-0} + 0 = 1/2 = 50\%$. Therefore, $F_1(u^n_{min}) > 50\%$. The proposed utilization bound $F_1(u^n_{min})$ for the uniprocessor slack-monotonic scheduling is higher than that of the state-of-the-art result in [And08b].

5.9 Summary

The preciseness of schedulability analysis for global FP scheduling is important in order to reduce the resource requirement by applying the corresponding schedulability test. Moreover, the efficiency of a schedulability test is also important in order to quickly determine if a task set is schedulable on a particular platform. Efficiency in evaluating a test enables the system designers to quickly apply the test for different choices of the parameters, e.g., different periods of each task, number of processors, and so on.

The density bound test for global FP scheduling based on the `ISM-DS` priority assignment policy is efficient: the total density can be computed in linear time and can be compared against the density bound in constant time. This test enables the designer to quickly determine, for a given number of processors, whether the timing constraints of a set of constrained-deadline sporadic tasks are met or not. In addition, the test also can be used to find the sufficient number of processors for meeting the timing constraints of a sporadic task set. The schedulability test using the `ISM-DS` priority assignment policy is proved to dominate the state-of-the-art density-bound test. The utilization bound test based on the priority assignment policy `ISM-DS` is also higher than other existing utilization bounds for global FP scheduling of implicit-deadline sporadic tasks. It is proved that the best possible utilization bound for scale invariant and independent priority assignment policy is achievable for `SM-US`$[\sqrt{2}-1]$ priority assignment policy if no task's utilization is in the range $(1-\frac{1}{\sqrt{2}}, \sqrt{2}-1]$. This test is highly effective for task sets with $m > 9$ and higher cardinality. The uniprocessor slack monotonic scheduling is shown to have a utilization bound higher that the state-of-the-art 50% untilization bound.

The priority assignment policy `ISM-DS`$[\xi]$ is derived based on the schedulability analysis of the global FP scheduling for the `ISM-DS` priority assignment policy. The schedulability test proposed for the `ISM-DS`$[\xi]$ priority assignment policy dominates the density-bound test proposed for global FP scheduling for the `ISM-DS` priority assignment policy. Searching the threshold density ξ from the set of densities of the tasks in a task set using algorithm `Find(`ξ`)`is efficient and can be done in $O(n \cdot \log n)$ time.

The simulation result shows significant improvement of the schedulability test for the `ISM-DS`$[\xi]$ priority assignment policy over the density-bound test proposed in this chapter. However, the performance of all the considered tests decreases as the cardinality increases for a given number of processors. This is because the total density of a task set increases with the increase in cardinality while the number of processors is fixed. The performance of the schedulability test for the `ISM-DS`$[\xi]$ priority assignment policy decreases if the number of processors increases for some fixed cardinality due to Dhall's effect. This is because the number of tasks having relatively larger individual density increases with the increase in number of processors while the number of tasks in a task set is fixed. In contrast, the density bound tests perform relatively better if the number of processors increases for a given cardinality of the task set since the average density of task set decreases in such case.

6
Iterative Tests

This chapter presents three new *iterative schedulability tests* for global FP scheduling of constrained-deadline sporadic task systems. Iterative schedulability test involves testing *one* schedulability condition for each task in a task set to determine whether its deadlines are met. One of the main challenges in deriving an iterative schedulability test is identifying the worst-case runtime behavior, i.e., called the critical instant. A job released at the critical instant suffers the maximum interference from the higher priority tasks. However, the critical instant is not yet known for global FP scheduling. To overcome this limitation, pessimism is introduced during the schedulability analysis to safely approximate the worst-case. The endeavor in this chapter is to reduce the different sources of pessimism in the state-of-the-art schedulability analysis and propose better iterative schedulability tests for global FP scheduling.

Another challenge for global FP scheduling is the problem of assigning the fixed priorities to the tasks since the optimal priority ordering in such case is still unknown. Each of the new schedulability tests proposed in this chapter combines the schedulability test for each task with finding its fixed priority using the principle of Audsley's priority assignment policy. Finding the priority assignments for all the tasks implies that the task set is schedulable using global FP scheduling. It is shown that the proposed tests dominate and empirically perform better than the state-of-the-art iterative schedulability test for constrained-deadline sporadic tasks.

6.1 Introduction

In many real-time systems, e.g., avionics, spacecraft and automotive, it is important to efficiently use the processing resources due to size, weight and power constraints. Reducing the resource requirement (e.g., number of processors) of such systems would significantly cut costs for mass production, for example, of cars, trucks or aircrafts. However, if the pessimism in the schedulability analysis for such systems is large, then a relatively higher number of processors is required to meet the deadlines. The endeavor in this chapter is to reduce such pessimism by proposing better iterative schedulability tests for global FP scheduling.

Global FP scheduling of constrained-deadline sporadic tasks systems is important not only for CPU scheduling but also in other domains, for example, scheduling real-time flows in WirelessHART networks designed for industrial process control and monitoring. WirelessHART is an open wireless sensor-actuator network standard specifically designed for industrial process control to avoid severe economic loss or environmental threats, reduce production inefficiency, enhance equipment monitoring and maintenance [WHA]. The analysis of global FP scheduling has been applied to the end-to-end delay analysis and priority assignment of the periodic real-time flow scheduling on multiple communication channels of WirelessHART networks [SXLC11a, SXLC11b]. Improvement of global FP schedulability analysis and the priority assignment policy would result in less pessimistic end-to-end delay calculation and would enhance the schedulability of the real-time flows transmitted over multiple communication channels in WirelessHART networks; and consequently, better control and monitoring of industrial processes can be attained.

Since the optimal priority assignment for global FP scheduling on a multiprocessor system (at present time) is unknown, the quality (e.g., minimum number of processors required) of many previously proposed global FP schedulability tests depends on the actual priority ordering of the tasks. Therefore, determining a good priority ordering is as important as deriving a good schedulability test. In this chapter, novel priority assignment schemes and the corresponding schedulability tests for scheduling such task systems on multiprocessors are proposed and demonstrated, using proof and simulation, that the schemes are superior to prior schemes.

Three new iterative schedulability tests for global FP scheduling are proposed: each test combines schedulability analysis of each task with priority assignment using Audsley's approach such that successful priority assignment implies the schedulability of the task. In other words, if all the tasks are assigned priorities using this combination, then the task set is also schedulable. Each of these iterative tests dominates the state-of-the-art iterative test for global FP scheduling of constrained-deadline sporadic tasks.

State-of-the-art Iterative Test. The basic idea of iterative schedulability test is that *one* condition is tested for each lower-priority task $\tau_i \in \Gamma$. The schedulability analysis of each task τ_i is performed within an interval, called the *problem window*, such that one job of the task τ_i is assumed to be released at the beginning of the problem window. One flavor of iterative test is based on computing the upper bound on the *response-time* of

task τ_i: the problem window size is initially set to C_i, then the response-time of task τ_i within the problem window is calculated; and, if the computed response-time of task τ_i is greater than the length of the problem window, then the size of the problem window is reset to the response-time just computed, and the process is repeated until the length of the problem window is not greater than the relative deadline of task τ_i. The iterative schedulability test proposed by Guan et al. in [GSYY09] for global FP scheduling, called the RTA-LC test, is the state-of-the-art response-time based iterative schedulability test[1]. The RTA-LC test derives an upper bound on the response time of each task τ_i using the response time of each higher priority tasks in set HP_i.

Another flavor of iterative test is based on *deadline-analysis* where the length of the problem window of task τ_i is set equal to its relative-deadline D_i and the schedulability analysis of task τ_i with this problem window is considered. In deadline-based analysis, an upper bound on the interference due to all the higher priority tasks on task τ_i in an interval of length D_i is computed. Then, based on the interference within the problem window, the minimum available time to execute task τ_i in the problem window is calculated. The iterative schedulability test proposed by Davis and Burns in [DB11b] for global FP scheduling, called the DA-LC test, is the state-of-the-art iterative schedulability test based on deadline-analysis.

It has been shown in [DB11b] that, for any *given* FP ordering of the tasks, the RTA-LC test dominates the DA-LC test. Nevertheless, the work in [DB11b] derives an effective joint priority assignment policy and schedulability test by combining the DA-LC test with multiprocessor extension of Audsley's optimal priority assignment (OPA) algorithm[2] [Aud01]. However, the RTA-LC test can not be combined with the OPA algorithm to find *another* priority ordering when the task set does not satisfy the RTA-LC test for the *given* priority assignment [DB11b]. It is empirically shown in [DB11b] that the combination of OPA and DA-LC test, called the ODA-LC test in this thesis, outperforms the RTA-LC test regardless of what heuristic priority assignment policy (e.g., deadline-monotonic) the latter uses. The ODA-LC test is the state-of-the-art iterative schedulability test for global FP scheduling of constrained-deadline sporadic tasks.

Contributions. The main contribution in this chapter is to identify the sources of pessimism in the analysis of state-of-the-art ODA-LC test and applying techniques to reduce such pessimism. In this chapter, three new iterative schedulability tests (each dominates the ODA-LC test) are proposed by increasingly improving the ODA-LC test. The overview of the main techniques for deriving the three tests is briefly presented below.

- **The H-ODA-LC Test:** This test combines the HPA policy with the ODA-LC test. Regarding the optimality of the ODA-LC test (as claimed in [DB11b]), it is observed that (i) optimality is only claimed under the assumption that the entire task set and all the processors are involved when the ODA-LC test is applied for determining the fixed-priority ordering of all the tasks, and (ii) the details of the

[1]The name "RTA-LC" test (response-time analysis with limited carry-in tasks) is introduced in [DB11b].
[2]The Audsley's OPA algorithm, adapted for multiprocessors, is presented in Section 6.2.1.

schedulability analysis of the ODA-LC test in [DB11b] imply that, if not all tasks and processors are included in the analysis, the upper bound on the interference due to the higher priority tasks on a lower priority task may be lowered. Based on this finding, the first new iterative schedulability test, called *HPA-applied ODA-LC* (H-ODA-LC) test, which dominates the ODA-LC test is proposed.

In H-ODA-LC test, at most m' largest-density tasks are given the highest fixed priorities and the remaining $(n - m')$ tasks are given other, lower, fixed priorities for some m', $0 \leq m' < m$. While the OPA algorithm is not (as shown in [DB11b]) applicable to the RTA-LC test, the HPA policy is indeed applicable to the RTA-LC test. The HPA policy combined with the RTA-LC test resulted in *HPA-applied RTA-LC* (H-RTA-LC) test which dominates the RTA-LC test.

- **The IA-DA Test:** The second contribution is proposing a novel idea to further improve the H-ODA-LC test. The purpose of assigning the highest fixed priorities to the m' largest-density tasks in the H-ODA-LC test is to reduce the pessimism involved in the interference computation of the higher priority tasks on a lower priority task. However, Observation 5.1 (page 58, Chapter 5) does not necessarily imply that the highest-density tasks are the best candidates for assigning the highest fixed priorities for the HPA-based priority assignment policy.

  It will be shown that it is not necessarily the highest-density tasks that may cause the maximum interference on a lower priority task. This crucial observation motivates the design of a new deadline-analysis-based iterative test, called the *Interference-Aware Deadline-Analysis* (IA-DA) test, for global FP scheduling of constrained–deadline sporadic tasks. A new *criterion* for identifying the tasks that are mostly responsible for pessimistic computation of interference on each lower-priority task is proposed. Based on this criterion, a novel priority-assignment technique, based on the principle of Audsley's OPA algorithm, is proposed. It is proved that if all the tasks are successfully assigned priorities using the proposed priority-assignment policy, then all deadlines of the tasks are met. It is also proved that the IA-DA test dominates the H-ODA-LC test.

- **The IA-RT Test:** It will be evident later that the IA-DA test essentially applies the deadline-based analysis to determine whether a task τ_i can be assigned (based on Audsley's algorithm) a particular priority level. While a deadline-based analysis considers a problem window of length D_i, a response-time based schedulability analysis considers a problem window smaller than D_i. And, the way the interference on a lower priority task is approximated for global FP scheduling (e.g., in DA-LC test) implies that a problem window larger than the response time of a the analyzed task is more pessimistic for interference computation.

  The IA-DA test is improved by considering a response-time based test[3] to determine whether a lower priority task τ_i can be assigned a particular priority level

[3]The response-time based test that will be used for IA-RT test is *not* the OPA-incompatible RTA-LC test; rather an OPA-compatible response-time-based test proposed in [DB10] is used.

based on the OPA algorithm. This new test is called *Interference-Aware Response-Time* (IA-RT) test which dominates the IA-DA test and significantly outperforms the state-of-the art ODA-LC test in simulation.

Organization. The rest of the chapter is organized as follows: Section 6.2 presents a schedulability analysis framework, an overview of Audsley's OPA algorithm and its applicability to multiprocessors. Section 6.3 presents the related works and the two state-of-the-art RTA-LC and ODA-LC iterative schedulability tests. The H-ODA-LC, IA-DA, IA-RT tests are presented in Sections 6.4 – 6.6, respectively. Simulation results are presented in Section 6.7 before summarizing the results in Section 6.8.

6.2 An Analysis Framework

In this section, an overview of the schedulability analysis framework to derive an iterative schedulability test of global FP scheduling is presented. The schedulability analysis of a generic job of a lower priority task τ_i in the problem window of task τ_i is considered. The iterative schedulability test of task τ_i is derived by computing the *workload*, *interfering workload*, *total interfering workload* and *interference* of the higher priority tasks within the problem window. Before techniques to compute these terms are presented, their definitions are formally presented.

Workload. The *workload* of a higher priority task τ_k within the problem window of task τ_i is the cumulative length of intervals during which task τ_k executes in that window. In [BCL09, BC07, GSYY09], the work done by a job of a higher-priority task τ_k is considered as "carry-in" work within the problem window of a lower-priority task τ_i if a job of task τ_k is released before the beginning of the window and executes (partially or fully) within the window. If a higher-priority task is considered to constitute carry-in work, then its worst-case interference on the lower-priority task is higher than that of its non-carry-in counterpart. In the remainder of this chapter, the higher priority task τ_i is called a "carry-in task" (CI) if it is considered to have carry-in work within the problem window of a lower priority task τ_k; otherwise, τ_i is called a "non-carry-in task" (NC).

Interfering Workload. The *interfering workload* of a higher priority task τ_k is the cumulative length of the intervals during which jobs of task τ_k execute and job of task τ_i is ready but not executing within the problem window of task τ_i. The CI and NC interfering workloads of each higher priority task τ_k are determined based on the upper bound on the CI and NC workloads of task τ_k within the problem window, respectively.

Total Interfering Workload. The *total interfering workload* is the sum of interfering workload of all the higher priority tasks within the problem window. It is proved by Guan et al. in [GSYY09] that there are at most $(m-1)$ carry-in tasks within the problem window of any lower priority task for global FP scheduling of constrained-deadline sporadic tasks. The total interfering workload is calculated by adding the CI interfering workloads of $(m-1)$ carry-in tasks and the NC interfering workloads of the remaining

higher priority tasks. The $(m-1)$ carry-in tasks from the set of higher priority tasks are selected such that the total interfering workload is maximized.

Interference. The *interference* on a job of task τ_i within the problem window is the cumulative length of the intervals during which the job of task τ_i within its problem window is ready but not executing. The interference of the higher priority tasks on task τ_i within the problem window is calculated based on total interfering workload. Once the interference of the higher priority tasks within a problem window calculated, the amount of available execution time for the lower priority task τ_i within the problem window can be determined. Finally, based on the available execution time of a lower priority task τ_i, sufficient schedulability test for task τ_i is derived.

In deadline-based analysis (e.g., DA-LC test), the length of the problem window is equal to D_i (i.e., the relative deadline of task τ_i). If the difference between D_i and the interference within a problem window of length D_i is not smaller than the execution time C_i of task τ_i, then task τ_i meets its deadline. On the other hand, the response-time based analysis (e.g., RTA-LC test) initially sets the length of the problem window to C_i. Then based on the interference within the current problem window, the response-time of task τ_i is calculated. If the response-time is greater than the length of the current problem window, the length of the problem window is incremented (a new problem window is considered), and this process continues until (i) the computed response time is greater than the deadline (deadline may be missed), or (ii) the computed response time is exactly equal to the length of the current problem window (deadline is met).

The iterative schedulability tests proposed in [BCL09, BC07, GSYY09] assumes that the priority ordering of the tasks is known before applying the test. However, there is a class of iterative schedulability test, called OPA-compatible tests, that are applicable not only for task sets with known priority ordering but also can be used to search for priority ordering combined with Audsley's OPA algorithm [Aud01]. Finding a priority ordering using OPA algorithm is important because the optimal fixed-priority ordering for global FP scheduling is not known. If a task set is not guaranteed to be schedulable for a given priority ordering, then to ensure the schedulability of the tasks for that given priority ordering it may require to increase the number of processors or even re-specification of the parameters of the tasks. Applying Audsley's OPA algorithm, combined with a schedulability test, could avoid such costly approach by finding another priority ordering for which the task set passes the schedulability test. The details of the Audsley's OPA algorithm and the conditions for a schedulability test to be OPA-compatible are presented next.

6.2.1 Audsley's OPA Algorithm

Audsley's OPA algorithm, originally proposed for uniprocessors in [Aud01], is extended by Davis and Burns for priority assignment in global FP multiprocessor scheduling [DB09]. All the proposed iterative schedulability tests (H-ODA-LC, IA-DA and IA-RT) in this chapter use the principle of Audsley's OPA algorithm for priority assign-

ment. In this subsection, the necessary conditions that must be satisfied for a schedulability test to be OPA-compatible are presented. Then, the pseudo-code for OPA algorithm is formally presented in Figure 6.1.

Andersson and Jonsson [AJ] concluded that Audsley's OPA algorithm can not be applied to determine the optimal priority ordering for global FP scheduling even if an exact schedulability test (e.g., exact feasibility test for periodic tasks is proposed by Cucu and Goossens in [CGG11]) were known. The basis for this conclusion by Andersson and Jonsson is the following observation for implicit-deadline tasks [AJ]:

> "*For fixed priority preemptive global multiprocessor scheduling, there exist task sets for which the response time of a task depends not only on T_i and C_i of its higher priority tasks, but also on the relative priority ordering of those tasks*."

However, this observation does not exclude the possibility of using Audsley's OPA algorithm for sufficient schedulability test of global multiprocessor scheduling as is first pointed out in [DB09]. With respect to the applicability of Audsley's OPA algorithm, Davis and Burns [DB09, DB11b] categorize a global FP schedulability test S as being either OPA-compatible or OPA-incompatible. An OPA-compatible test S implies that Audsley's OPA algorithm can be applied to find priority assignment using test S. The clause "using test S" in the last sentence is very critical and also the basis for claiming the optimality of the priority assignment according to the combination of the schedulability test S and the OPA algorithm. **If an OPA-compatible test S can not find a priority ordering using the combination of OPA algorithm and the schedulability test S for a task set, it does not necessarily imply that there is no priority ordering for which the task set is global FP schedulable**. The adjective "optimal" in finding a priority ordering of a task set, based on the OPA algorithm and an OPA-compatible test S, must not lead to the following confusion:

> *The optimal fixed-priority assignment for global multiprocessor scheduling (an exciting and important result) is now known.*

Applying the OPA algorithm using an OPA-compatible test S essentially finds an optimal priority ordering only with respect to test S: *if a task set satisfies an OPA-compatible schedulability test S for some priority ordering, then that OPA-compatible test S can find such a priority ordering using the OPA algorithm.*

Conditions for OPA-Compatibility (from [DB09, DB11b])

A schedulability test S for global FP scheduling is OPA-compatible if the following three conditions are satisfied:

- **Condition 1:** The schedulability of a task τ_i may, according to test S, be dependent on the set of higher priority tasks, but not on the relative priority ordering of those tasks.

- **Condition 2:** The schedulability of a task τ_i may, according to test S, be dependent on the set of lower priority tasks, but not on the relative priority ordering of those tasks.

- **Condition 3:** When the priorities of any two tasks of adjacent priority are swapped, the task being assigned the higher priority can not become unschedulable according to test S, if it was previously schedulable at the lower priority. (As a corollary, the task being assigned the lower priority can not become schedulable according to test S, if it was previously unschedulable at the higher priority).

Audsley's OPA Algorithm for Multiprocessors

The OPA algorithm given in Figure 6.1 assigns fixed priorities to the tasks in set A to be scheduled on $\hat{m}$ processors based on some global FP schedulability test S that is OPA-compatible. Unlike the representation in [DB09, DB11b], the parameters (task set A, number of processors $\hat{m}$ and the OPA-compatible test S) of the OPA algorithm are made explicit here.

Algorithm OPA(Task set A, number of processors $\hat{m}$, Test S)

```
1.  for each priority level PL, lowest first
2.      for each priority-unassigned task τ ∈ A
3.          If τ is schedulable on m̂ processors at priority level PL
4.              according to schedulability test S with all other priority-
5.              unassigned tasks assumed to have higher priorities, Then
6.                  assign τ to priority PL
7.                  break (continue outer loop)
8.      return "failure"
9.  return "success"
```

Figure 6.1: *Audsley's OPA algorithm for multiprocessors.*

The OPA algorithm assigns priority to each task in set A starting from the lowest-priority level. In order to be used, the FP schedulability test S has to be OPA-compatible (i.e., needs to satisfy Conditions 1–3 given above). If the function call OPA(Γ, m, S) returns "success", then all deadlines of the tasks in Γ are met on m processors according to the priorities assigned by the OPA algorithm in Figure 6.1. Initially, all the tasks in set A are priority-unassigned. The objective of the OPA algorithm is to assign priority to each of the tasks in set A starting from the lowest priority level (i.e., the lowest priority task is determine first and the highest priority task is determined last).

The for loop in line 1 iterates for each of the priority level, denoted by PL, starting from the lowest priority level. For each priority level in line 1, one priority-unassigned task is searched using the inner loop in line 2 for assigning the priority at that priority level. Whether or not a (priority-unassigned) task, say task τ, can be assigned the particular priority level PL is determined in line 3–5 by applying the test S and assuming

the higher priorities for all other (priority-unassigned) tasks. If such a task τ is found, then that task is assigned the current priority level and the priority assignment for next higher priority level starts (starting from the outer loop).

If no task can be assigned the current priority level, the inner loop terminates and line 8 returns "failure". If the outer loop terminates after assigning priorities for each of the tasks in set A, then the algorithm returns "success". The OPA algorithm performs at most $n(n+1)/2$ schedulability tests in contrast to exhaustively applying the test for $n!$ different fixed-priority orderings of the tasks. The following theorem guarantees that algorithm OPA in Figure 6.1 always finds a priority assignment of the tasks if there exists some priority ordering that makes the task set to satisfy the schedulability test S.

Theorem 6.1 (from [DB09])**.** *The Optimal Priority Assignment (OPA) algorithm is an optimal priority assignment policy for any global* FP *schedulability test S compliant with Conditions 1-3.*

While Theorem 6.1 is undoubtedly true, it is *not* correct to say that if algorithm OPA in Figure 6.1 can not find a priority ordering using the OPA-compatible schedulability test S, then there is no other priority ordering that can make the task set schedulable.

6.3 Related Work

Several iterative tests are already been proposed in the literature for global FP scheduling of constrained-deadline sporadic tasks [Bak06, BC07, BCL09, GSYY09, DB11b]. A recent survey by Davis and Burns of different schedulability tests for global FP scheduling can be found in [DB11a]. Empirical investigations in [Bak06, BCL09, DB11b] show that such tests are highly effective in determining the schedulability of task sets having a total density / utilization beyond the state-of-the-art bound for implicit- / constrained-deadline tasks.

The basis of the schedulability analysis in many iterative tests is determining the interference on each lower priority task due to its higher priority tasks within a problem window. However, unlike the uniprocessor FP scheduling, the exact interference calculation for multiprocessor FP scheduling is difficult since the critical instant for global FP scheduling of sporadic tasks is not known (please see section 3.1). Consequently, an upper bound on the interference of the higher priority tasks on each lower priority task with the problem window is calculated to derive a sufficient schedulability test. Based on Baker's seminal work in [Bak06], several works [BCL09, BC07, GSYY09] have proposed iterative schedulability tests for constrained-deadline sporadic task systems based on bounding the amount of interference due to each of the higher priority tasks within the problem window of a lower priority task.

Many global FP schedulability analysis of a lower-priority task τ_i considers that all the higher-priority tasks to have carry-in work within the problem window [BCL09, BC07]. Baruah's global EDF schedulability analysis in [Bar07] limits the number of higher-priority tasks considered to have carry-in work to $(m-1)$, where m is the number of processors. The `RTA-LC` test proposed by Guan et al. [GSYY09] employs the

same carry-in task limitation as the analysis in [Bar07] to improve the response-time analysis proposed in [BC07] for global FP scheduling of constrained-deadline sporadic tasks. The test in [GSYY09] computes the upper bound on the response time of a task based on the response time of the higher priority tasks. Recently, inspired by the works in [BC07, Bar07, GSYY09], Davis and Burns [DB11b] proposed a test that also considers $(m-1)$ tasks having carry-in work to improve the deadline-based schedulability analysis in [BCL09] for global FP scheduling of constrained-deadline sporadic tasks. This improved test proposed by Davis et al. in [DB11b] is called DA-LC test (deadline-analysis with limited carry-in).

The RTA-LC test dominates the DA-LC test for any given fixed-priority ordering of the constrained-deadline tasks [DB11b]. However, Davis et al. [DB09, DB11b] addressed the problem of finding an effective priority assignment using Audsley's OPA algorithm [Aud01] for the class of schedulability tests that are OPA-compatible. To that end, RTA-LC is proved not to be OPA-compatible while DA-LC is proved to be OPA-compatible [DB11b]. It is empirically shown that OPA combined with DA-LC tests (i.e., the ODA-LC test) is currently the best combination of priority-assignment policy and schedulability test for global FP scheduling [DB11b]. The state-of-the-art response-time based RTA-LC test and deadline-based ODA-LC test are now presented in Subsection 6.3.1 in details to identify the pessimism in their schedulability analysis and to propose the H-ODA-LC, IA-DA and IA-RT tests in Sections 6.4 – 6.6, respectively.

6.3.1 State-of-the-art Iterative Tests

The RTA-LC is the response-time-based test and the DA-LC test is a deadline-analysis-based test. The RTA-LC test calculates an upper bound on the response time of each task. The response time of task τ_i determined using the RTA-LC test is denoted by R_i. Remember that HP_i is the set of all the higher-priority tasks of task τ_i. In order to understand the RTA-LC and DA-LC tests, we need to know how the workload, interfering workload, total interfering work, and interference within the problem window of any job of a lower priority task τ_i are calculated in [GSYY09] and [DB11b], respectively. The following equations Eq. (6.1) – (6.9) are presented in a different form than that are used in [GSYY09, DB11b] in order to show the similarities and differences between the DA-LC and RTA-LC tests.

Workload. There are at most $(m-1)$ tasks with carry-in workload within the problem window of each lower priority task τ_i in global FP scheduling [GSYY09]. Whether task $\tau_k \in \mathsf{HP}_i$ is a CI task or a NC task depends on the CI and NC workload of that task in the problem window. The upper bound on the workloads of task $\tau_k \in \mathsf{HP}_i$ within any interval of length t is denoted by $\mathsf{W}_k^{\mathsf{NC}}(t)$ and $\mathsf{W}_k^{\mathsf{CI}}(t)$ whenever τ_k is a NC task and CI task, respectively. The NC workload $\mathsf{W}_k^{\mathsf{NC}}(t)$ of task τ_k for both RTA-LC and DA-LC tests is given as follows [GSYY09, DB11b]:

$$\mathsf{W}_k^{\mathsf{NC}}(t) = \lfloor t/T_k \rfloor \cdot C_k + min(C_k, t - \lfloor t/T_k \rfloor \cdot T_k) \tag{6.1}$$

However, the CI workload for the RTA-LC test and the DA-LC tests are computed differently. The value of CI workload $\mathtt{W}_k^{\mathtt{CI}}(t)$ of task τ_k in an interval of length t for the RTA-LC test is given as follows [GSYY09]:

$$\mathtt{W}_k^{\mathtt{CI}}(t) = A_t^k \cdot C_k + min(C_k, t + R_k - C_k - A_t^k \cdot T_k) \tag{6.2}$$

where $A_t^k = \lfloor (t + R_k - C_k)/T_k \rfloor$. Note that R_k is an upper bound on the response time of the higher priority task $\tau_k \in \mathtt{HP}_i$ and R_k has to be calculated before R_i is calculated. The dependence on the response time of the higher priority task τ_k when calculating the CI workload $\mathtt{W}_k^{\mathtt{CI}}(t)$ for analyzing the schedulability of lower-priority task τ_i makes the RTA-LC test OPA-incompatible. This is because the response-time of higher priority task τ_k depends on the relative priority ordering of the task in $\mathtt{HP}_i$ (violates Condition 1 given in page 83). The value of CI workload $\mathtt{W}_k^{\mathtt{CI}}(t)$ of task τ_k in an interval of length t for the DA-LC test is given as follows [DB11b]:

$$\mathtt{W}_k^{\mathtt{CI}}(t) = A_t^k \cdot C_k + min(C_k, t + D_k - C_k - A_t^k \cdot T_k) \tag{6.3}$$

where $A_t^k = \lfloor (t + D_k - C_k)/T_k \rfloor$. Given the length of the problem window t, the value of $\mathtt{W}_k^{\mathtt{CI}}(t)$ for the DA-LC test is calculated only using the static parameters[4] of task τ_k.

Interfering Workload: Similar to workload, $\mathtt{I}_{k,i}^{\mathtt{CI}}(t)$ and $\mathtt{I}_{k,i}^{\mathtt{NC}}(t)$ denote the upper bounds on the interfering workload of task τ_k on any job of task τ_i within the problem window of length t whenever τ_k is a CI task and NC task, respectively. An upper bound on the interfering workload of a higher priority task within the problem window is the workload of the higher priority task within that problem window. However, it is pointed out in [BC07, GSYY09, DB11b] that it is sufficient to consider the interfering workload of a higher priority task limited to at most $(t - C_i + 1)$ within the problem window size t. Thus, $\mathtt{I}_{k,i}^{\mathtt{CI}}(t)$ and $\mathtt{I}_{k,i}^{\mathtt{NC}}(t)$ for both DA-LC and RTA-LC tests are given as follows:

$$\mathtt{I}_{k,i}^{\mathtt{CI}}(t) \ = min(\mathtt{W}_k^{\mathtt{CI}}(t), t - C_i + 1) \tag{6.4}$$

$$\mathtt{I}_{k,i}^{\mathtt{NC}}(t) \ = min(\mathtt{W}_k^{\mathtt{NC}}(t), t - C_i + 1) \tag{6.5}$$

The CI interfering workload of higher priority task τ_k is never smaller than its NC interfering workload. In other words, $\mathtt{I}_{k,i}^{\mathtt{CI}}(t) \ \geq \mathtt{I}_{k,i}^{\mathtt{NC}}(t)$. The difference between the CI and NC interfering workload of task τ_k within the problem window of length t is denoted by $\mathtt{I}_{k,i}^{\mathtt{DIFF}}(t)$ and given as follows:

$$\mathtt{I}_{k,i}^{\mathtt{DIFF}}(t) \ = \mathtt{I}_{k,i}^{\mathtt{CI}}(t) \ - \mathtt{I}_{k,i}^{\mathtt{NC}}(t) \tag{6.6}$$

The value of $\mathtt{I}_{k,i}^{\mathtt{DIFF}}(t)$ determines whether the higher priority task τ_k has to be considered as a CI task or NC task within the problem window of length t.

Total Interfering Workload. The upper bound on total interfering workload on task τ_i due to all the higher priority tasks in set ψ is denoted as $\mathtt{I}_i(t, \psi, m)$; where $\psi \subseteq \mathtt{HP}_i$,

[4]The static parameters describe characteristics of a task that apply independent of other tasks.

the length of the problem window is t, and the tasks are scheduled on m processors. Total interfering workload $\mathtt{I}_i(t, \psi, m)$ is the sum of the interfering workload of all tasks in set ψ where at most $(m-1)$ tasks are considered as $\mathtt{CI}$ tasks. The $(m-1)$ carry-in tasks from set ψ are those tasks that have the largest value of $\mathtt{I}^{\mathtt{DIFF}}_{k,i}(t)$. The value of $\mathtt{I}_i(t, \psi, m)$ is calculated as follows for both the DA-LC and RTA-LC tests:

$$\mathtt{I}_i(t, \psi, m) = \sum_{\tau_k \in \psi} \mathtt{I}^{\mathtt{NC}}_{k,i}(t) + \sum_{\tau_k \in Max(\psi, m-1)} \mathtt{I}^{\mathtt{DIFF}}_{k,i}(t) \tag{6.7}$$

where $Max(\psi, m-1)$ is the set of $(m-1)$ tasks from set ψ that have the largest values of $\mathtt{I}^{\mathtt{DIFF}}_{k,i}(t)$.

Interference. The term interference is an integer and all the m processors are busy executing tasks from ψ while task τ_i is interfered by the higher priority tasks in $\psi \subseteq \mathsf{HP}_i$. Thus, based on the schedulability analysis in [BC07, GSYY09, DB11b], an upper bound on interference due to the tasks in ψ on any job of task τ_i within the problem window of length t is $\lfloor \frac{\mathtt{I}_i(t,\psi,m)}{m} \rfloor$.

The RTA-LC test: The RTA-LC test [GSYY09], which computes an upper bound on the response time of each lower priority task $\tau_i \in \Gamma$, is recursively given as follows:

$$R_i^{(h+1)} \leftarrow C_i + \left\lfloor \frac{\mathtt{I}_i(R_i^h, \mathsf{HP}_i, m)}{m} \right\rfloor \tag{6.8}$$

This can be solved by searching iteratively the least fixed point starting with $R_i^0 = C_i$ for the right-hand side of Eq. (6.8). Thus, this recursion starts with $R_i^0 = C_i$ and stops when either (i) $R_i^{(h+1)} > D_i$ (i.e., task τ_i can not be guaranteed schedulable) or (ii) $R_i^{h+1} = R_i^h$ (i.e., task τ_i is schedulable with response time $R_i = R_i^{h+1}$). Note that in order compute the response time of task τ_i using Eq. (6.8), the response time of each higher priority task $\tau_k \in \mathsf{HP}_i$ must be known. It is not difficult to see that the computational complexity of the RTA-LC test is pseudo-polynomial and the dependency on knowing the response time of the higher priority task $\tau_k \in \mathsf{HP}_i$ to compute the response time of task τ_i makes the RTA-LC test OPA-incompatible.

DA-LC Test: The DA-LC test [DB11b] for each lower priority task $\tau_i \in \Gamma$ with relative deadline $D_i \leq T_i$ is given as follows:

$$D_i \geq C_i + \left\lfloor \frac{I_i(D_i, \mathsf{HP}_i, m)}{m} \right\rfloor \tag{6.9}$$

This can be solved by calculating the interference of the higher priority tasks in HP_i within the problem window of length D_i. It is not difficult to see that the computational complexity of the DA-LC test is polynomial and the test is OPA-compatible.

ODA-LC Test: The DA-LC test is OPA-compatible and can be used to find the FP ordering of the tasks using the Audsley's OPA algorithm presented in Figure 6.1. The

ODA-LC test (combination of OPA and DA-LC test) works as follows [DB11b]:

> If the call OPA($\Gamma, m,$ DA-LC) in Figure 6.1 returns "success", then all the tasks meet deadlines using global FP scheduling on m processors based on the priority assignment determined by the OPA algorithm.

According to the ODA-LC test, the OPA algorithm in Figure 6.1 essentially applies the DA-LC test in Eq. (6.9) to determine whether a priority-unassigned task can be assigned a particular priority level by assuming the higher priorities for all the other priority-unassigned tasks. It will be evident shortly that, the H-ODA-LC test (proposed in next section) is based on applying the HPA policy where not all the higher priority tasks and all the m processors are considered when determining the priority level of a priority-unassigned task based on the DA-LC test.

6.4 The H-ODA-LC Test

In this section, the HPA policy is applied to improve the priority assignment policies for two state-of-the-art iterative schedulability tests: ODA-LC test proposed by Davis et al. [DB11b] and OPA-incompatible RTA-LC test proposed by Guan et al. [GSYY09].

The OPA algorithm in Figure 6.1 reveals an interesting fact: the priority-assignment determined by the combination of the OPA algorithm and an OPA-compatible schedulability test S only claims to be optimal under the assumption that this combination is applied to the *entire* task set and to *all* processors (Theorem 3 in [DB11b]). An intuitive question to ask is then whether it would be possible to obtain a more effective priority assignment for an OPA-compatible test if the combination of the OPA algorithm and the OPA-compatible test was applied to find the priorities of a subset of the entire task set to be scheduled on a lower number of processors while the remaining tasks are assigned fixed priorities using some other mechanism (e.g., the highest fixed priority as is proposed for ISM-DS policy in Chapter 5). By carefully studying the equations of the OPA-compatible DA-LC test presented in subsection 6.3.1, it is realized that there is indeed room for improvement.

In this section, the HPA policy is considered to improve the priority assignment policy for the ODA-LC test. This is based on a crucial observation: **the amount of interference calculated based on the DA-LC test on a lower priority task can be reduced by not including all the tasks and all the processors in the schedulability test.** The HPA policy combined with the ODA-LC test is called the H-ODA-LC test. Moreover, the HPA policy can also be applied to the OPA-incompatible RTA-LC test. The HPA policy combined with the RTA-LC test is called the H-RTA-LC test which dominates the RTA-LC test.

6.4.1 Applying HPA Policy to ODA-LC Test

In this subsection, by applying the HPA policy to the ODA-LC test an improved fixed-priority assignment policy and the schedulability test, called H-ODA-LC test, is pro-

posed. When computing the total interfering workload $I_i(D_i, \mathsf{HP}_i, m)$ in Eq. (6.9), for testing the schedulability of the lower priority task τ_i on m processors using the DA-LC test, the higher priority tasks are in HP_i and the number of CI tasks considered is $(m-1)$. The improved priority-assignment policy H-ODA-LC is based on the following observation of Eq (6.9): if one task, say τ_h, is removed from HP_i and also the number of processors is reduced from m to $(m-1)$, and apply the DA-LC test on this smaller task set and reduced number of processors, then the interference on task τ_i depends on the the higher priority tasks in set $(\mathsf{HP}_i - \{\tau_h\})$ and on $(m-2)$ carry-in tasks. To understand the importance of this observation, consider the following example.

Example 6.1. Consider four tasks in $\Gamma = \{\tau_1, \ldots \tau_4\}$ to be scheduled on $m = 3$ processors using global FP scheduling. The parameters (C_i, D_i, T_i) of the four tasks are as follows: $(23, 33, 33)$, $(106, 210, 214)$, $(58, 216, 217)$, and $(46, 60, 64)$. The ODA-LC test by calling algorithm OPA(Γ,3,DA-LC) returns "failure" because no task in Γ can be assigned the lowest priority level. This is because, when the schedulability of each $\tau_i \in \Gamma$ is checked for priority assignment as the lowest priority level (line 3-5 of OPA algorithm in Figure 6.1), the calculation of $I_i(D_i, \mathsf{HP}_i, m)$ using Eq. (6.7) considers $(m-1) = 2$ tasks in HP_i as CI tasks and the remaining task in HP_i as NC tasks. The value of $I_i(D_i, \mathsf{HP}_i, m)$ for each of the four tasks was large (pessimistic) enough to violate the DA-LC test in Eq. (6.9), and no task is decided to be assigned the lowest priority and the OPA algorithm returns "failure".

Now consider hybrid-priority assignment in which the highest-density task τ_4 is given the highest fixed priority. The call OPA($\{\tau_1, \tau_2, \tau_3\}$, 2, DA-LC) by removing τ_4 from Γ and reducing the number of processors from $m = 3$ to $m = 2$ returns "success" (task τ_3 is assigned the lowest priority, tasks τ_1 and τ_2 are assigned the highest fixed priorities). Therefore, the task set Γ is schedulable on $m = 3$ processors (follows from Observation 5.1 following Lemma 5.6). This is because, when OPA($\{\tau_1, \tau_2, \tau_3\}$, 2, DA-LC) is called, the calculation of $I_3(D_3, \{\tau_1, \tau_2\}, m = 2)$ in Eq. (6.7) considers only $(m-1) = 1$ task in $\{\tau_1, \tau_2\}$ as CI task and one task in $\{\tau_1, \tau_2\}$ as NC task. In this case, $I_3(D_3, \{\tau_1, \tau_2\}, m = 2)$ was small enough to satisfy the DA-LC test in Eq. (6.9) and τ_3 is assigned the lowest priority. The other two tasks, τ_1 and τ_2, are trivially assigned the highest fixed priority since there are two processors. Hence, OPA($\{\tau_1, \tau_2, \tau_3\}$, 2, DA-LC) returns "success". Since τ_4 is assigned the highest fixed priority and OPA($\{\tau_1, \tau_2\tau_3\}$, 2, DA-LC) returns "success", this instance of HPA guarantees that Γ is schedulable on $m = 3$ processors (from Observation 5.1). □

The important conclusion from this example is that, **if the schedulability of Γ can not be decided on m processors by applying the ODA-LC test to the *entire* task set Γ and to *all* m processors, it does not necessarily mean that there is no feasible priority assignment for Γ based on the DA-LC test.** The lesson learned is that the upper bound on interference $I_i(D_i, \mathsf{HP}_i, m)$, calculated based on DA-LC using Eq. (6.7), may be lowered by not including all the tasks and all the processors in the corresponding schedulability test. The HPA policy can exploit this because it provides "separation of concerns" in the sense that (i) the ODA-LC test can be applied (due to the predictability

of global FP scheduling) only to the $(n - m')$ lowest-density tasks to be scheduled on $(m - m')$ processors, and (ii) the remaining m' highest-density tasks are assigned (without any concern) the highest fixed priorities for some m', $0 \leq m' < m$. This is the main principle in developing the improved H-ODA-LC test.

Based on Observation 5.1, the entire task set Γ is global FP schedulable if the $(n - m')$ lowest-density tasks are schedulable using the ODA-LC test on $(m - m')$ processors. Note that the H-ODA-LC test dominates the ODA-LC test (i.e., when $m' = 0$, H-ODA-LC is equivalent to the ODA-LC test; and Example 1 shows the superiority of H-ODA-LC to the ODA-LC test). Figure 6.2 shows the pseudocode for the H-ODA-LC test. Each of the m' highest density tasks is assigned the highest fixed priority in line 4 of Figure 6.2 and the remaining $(n - m')$ tasks are tested for schedulability using the ODA-LC test on $(m - m')$ processors in line 6. If the OPA returns "success" (in line 6) for some m', $0 \leq m' < m$, then the task set Γ is decided to be FP schedulable.

Algorithm H-ODA-LC(Γ , m)

1. for $m' = 0$ to $(m - 1)$
2. if$(m' > 0)$ then
3. $\tau_h \leftarrow$ the highest-density task in Γ
4. assign τ_h the highest fixed priority
5. $\Gamma = \Gamma - \{\tau_h\}$ // one task is removed
6. if OPA(Γ, $m - m'$, DA-LC) returns "success" then
7. return "schedulable"
8. return "schedulability can not be determined" // when the for loop ends

Figure 6.2: *The H-ODA-LC test*

Remember that the OPA algorithm can not be applied to the RTA-LC test since it is OPA-incompatible [DB11b]. However, HPA policy is applicable to the RTA-LC test as follows (called, the H-RTA-LC test): *assign the m' highest-density tasks the highest fixed priorities and the fixed-priority ordering of the remaining $(n - m')$ lowest-density tasks remains the same as the original fixed-priority ordering that is given for the entire task set* Γ. Using Observation 5.1 following Lemma 5.6, the entire task set Γ is global FP schedulable if the $(n - m')$ lowest-density tasks are feasible using the RTA-LC test on $(m - m')$ processors for some m', $0 \leq m' < m$. For a given priority assignment for Γ, it is not hard to see that the H-RTA-LC test dominates the RTA-LC test.

It is empirically shown in [DB11b] that the ODA-LC test significantly performs better that the RTA-LC test. Therefore, it is expected that the H-ODA-LC test also guarantees such improvement over the H-RTA-LC test. The IA-DA schedulability test proposed in next section further improves the H-ODA-LC test.

6.5 The IA-DA Test

A new priority assignment policy and schedulability test, called the IA-DA test, is proposed in this section. The H-ODA-LC test in Section 6.4 is developed by observing that if not all the higher-priority tasks and all the processors are included when applying the DA-LC test to a lower-priority task, the pessimism in the estimation of the upper bound on interference due to the higher-priority tasks on a lower priority task can be reduced. The basic idea for applying the HPA policy in H-ODA-LC test is to keep some tasks and processors "separate" from the schedulability analysis of a lower priority task. Notice that the H-ODA-LC test "separates" a total of m' highest-density tasks, here referred to as "**separated tasks**", and "separates" a total of m' processors, here referred to as "**separated processors**", from the schedulability analysis of the remaining $(n - m')$ lowest-density tasks. The separated tasks and processors are not considered while evaluating the DA-LC test for a lower-priority task. Therefore, *the number of CI tasks when applying the DA-LC test to each of the* $(n-m')$ *lower-priority tasks in the ODA-LC test is limited to at most* $(m - m' - 1)$ *rather than* $(m - 1)$ *for some* m', $0 \leq m' < m$.

In this section, a new and novel criterion is proposed to determine the set of tasks that are separated when analyzing the schedulability of a lower-priority task. The proposed criterion for separating tasks is special in the sense that it is ***not*** based on "highest density" and separates ***different*** set of tasks for each lower priority tasks. The "separation" of tasks and processors has nothing to do with partitioned multiprocessor scheduling — the separation only exists as a means for reducing the pessimism of interference due to the higher-priority tasks on a lower-priority task.

Based on this new criterion, a new priority-assignment algorithm and the corresponding IA-DA test for global FP scheduling is presented. First, an overview of the proposed priority-assignment policy is presented in subsection 6.5.1. Then, in subsection 6.5.2, the elegant criterion for finding the set of separated tasks for a lower-priority task is proposed. Finally, the algorithmic details of the priority-assignment policy and the IA-DA test based on this new criterion is proposed in subsection 6.5.3.

6.5.1 Overview of the IA-DA Test

In this subsection, an overview of the priority assignment for the IA-DA test is presented. The IA-DA test checks whether all the tasks are successfully assigned priorities while at the same time also verifies the schedulability of the tasks. If all the tasks are assigned priority, then it is also guaranteed that all the tasks meet their deadlines.

The proposed priority-assignment policy applies the principle of Audsley's OPA algorithm: it assigns priorities to the tasks starting[5] from lowest-priority level PL=1 to the highest priority level PL=n. At each priority level PL, all tasks that are not yet assigned any priority are called the *priority-unassigned tasks*. The objective is to assign

[5] In this chapter, it is assumed without loss of generality that a task having priority level 1 (n) has the lowest (highest) fixed priority. This simplifies the mathematical reasoning in proving the correctness and domination of the IA-DA test.

fixed priority to one of the priority-unassigned tasks at each priority level. Each of the priority-unassigned tasks at each priority level is checked for priority assignment using the DA-LC test until one such task satisfying the DA-LC test is found.

Each of the priority-unassigned tasks when selected as a candidate for priority assignment is called the ***target*** task. Given a target task at priority level PL, the IA-DA test *temporarily* separates m' processors and separates m' tasks from the set of other priority-unassigned tasks where $0 \leq m' < m$. Unlike the previously proposed H-ODA-LC test, the m' separated tasks are *not* assigned any priority when separated, and more importantly, the criterion for selecting the separated tasks is *not* based on the "highest density". A new criterion for selecting the tasks for separation for each target task at each priority level is proposed (the criterion will be presented in Subsection 6.5.2).

After separating m' tasks for a particular target task at priority level PL, it is checked (using the DA-LC test in Eq. (6.9)) whether or not the target task can be assigned priority level PL. The separated tasks and separated processors are not considered while evaluating the DA-LC test for the target task. If the target task passes the DA-LC test at priority level PL, then the task is assigned priority level PL. If the target task does not pass the DA-LC test at priority level PL, then another priority-unassigned task is selected as the target for priority assignment at priority level PL. If no priority-unassigned task can be assigned priority level PL, the priority assignment *fails*. If all tasks are assigned priorities, then the priority assignment *succeeds*.

When a target task can not be assigned priority level PL, the corresponding separated tasks and separated processors are no more considered "separated". These tasks along with other priority-unassigned tasks are considered as candidates for selecting the next target task at priority level PL. Similarly, if a target task is assigned priority level PL, then the corresponding separated tasks and separated processors are no more considered "separated". And, these tasks are also considered as candidates for target task at next priority level. Thus, the separated tasks and separated processors for each target task are temporary in the sense that *priority assignment for each new target task always starts with all the m processors and all the priority-unassigned tasks*.

6.5.2 New Criterion for Separation

In this subsection, the elegant criterion for separating the tasks for each target task τ_i is designed. Remember that H-ODA-LC test separates m' *highest-density* tasks from Γ and then applies the ODA-LC test to the remaining $(n - m')$ lowest density tasks using $(m - m')$ processors for some m', $0 \leq m' < m$. Note that the *same* set of m' highest-density tasks having the highest fixed priorities are always kept separated from all the $(n - m')$ lowest-density tasks in H-ODA-LC test. These separated m' highest-density tasks are "constant" in the sense that the same set of m' highest density tasks are kept separated when determining the priorities of the $(n - m')$ lowest-density tasks on $(m - m')$ processors based on the ODA-LC test.

The reason for separating the highest density tasks in H-ODA-LC test is the feelings that the tasks that are responsible the most, for the pessimism involved in the interfer-

ence calculation using the DA-LC test applied to task τ_i, are the highest-density tasks. However, by studying the details of the proposed H-ODA-LC test, a very interesting fact is observed: **it is not necessarily the pessimism of the interference estimation due to the highest-density tasks that may cause some lower priority task τ_i to fail the DA-LC test**. To see why, consider the following example:

Example 6.2. Consider four tasks in $\Gamma = \{\tau_1, \ldots \tau_4\}$ to be scheduled on $m = 3$ processors using global FP scheduling. The parameters (C_i, D_i, T_i) of the four tasks are as follows: $(26, 51, 54)$, $(11, 14, 25)$, $(32, 33, 37)$, and $(19, 25, 29)$. The densities are $\delta_1 = 0.509$, $\delta_2 = 0.785$, $\delta_3 = 0.967$, and $\delta_4 = 0.760$. The task set Γ does not pass the H-ODA-LC test. In particular, none of the tasks in Γ can be assigned the lowest priority level by separating m' highest-density tasks for any $m' = 0, 1, 2$.

However, there exits a valid fixed priority assignment that would make task set Γ global FP schedulable. Consider that the two tasks $\{\tau_3, \tau_4\}$ are separated along with $m' = 2$ processors. The other two tasks $\{\tau_1, \tau_2\}$ are schedulable on $(m - m') = 1$ processor by assigning the two lowest priority levels PL=1 and PL=2 to tasks τ_1 and τ_2, respectively. Then, the two separated tasks τ_3 and τ_4 are assigned the highest priority levels PL=3 and PL=4, respectively. These two highest priority tasks τ_3 and τ_4 are trivially schedulable since we have $m = 3$ processors; and these two highest priority tasks uses at most two processors at any time. Evidently, at least one processor is always available for executing the two lowest priority tasks τ_1 and τ_2. Consequently, the entire task set is global FP schedulable based on observation 5.1. ***Note that the two separated tasks τ_3 and τ_4 are not the two highest density tasks.*** □

The lesson learned is that "separation" based on the HPA policy is effective; however, the best criterion to separate the tasks from the schedulability analysis of the lower priority tasks is not necessarily should be based on "highest density". Another important fact is that the (*constant*) set of m' highest-density tasks may not be the *best* set of separated tasks when checking the schedulability for each of the lower-priority tasks using the DA-LC test. A new criterion for separating the tasks when considering the priority assignment of a target task using the DA-LC test is proposed for this purpose. As will be evident now *the proposed criterion separates different sets of tasks for each possible target task at each priority level*.

Proposed Separation Criterion

Consider a target task τ_i at priority level PL where HP_i is the set of all the higher-priority tasks of τ_i. Assume that task τ_i does not pass the DA-LC test when applying the DA-LC test by considering all the tasks from HP_i and all the m processors. So, according to Eq. (6.9), the upper bound on interference, i.e., $\lfloor \frac{I_i(D_i, \mathsf{HP}_i, m)}{m} \rfloor$, that task τ_i suffers due to the tasks in HP_i is greater than $(D_i - C_i)$.

Now, separating m' tasks from set HP_i and separating m' processors may able task τ_i to pass the DA-LC test. The objective is to separate those m' tasks from HP_i such that the interference $\lfloor \frac{I_i(D_i, \mathsf{HP}_i, m)}{m} \rfloor$ is *maximally* reduced. And, m' processors

are also kept separated in such case. If SEP is the set of m' separated tasks selected from set $\mathtt{HP}_i$, then the value of (new) interference on any job of task τ_i (after separation) is $\lfloor \frac{I_i(D_i, \mathtt{HP}_i - \mathtt{SEP}, m-m')}{m-m'} \rfloor$ where the computation of total interfering workload $I_i(D_i, \mathtt{HP}_i - \mathtt{SEP}, m-m')$ considers $(m-m'-1)$ carry-in tasks from set $(\mathtt{HP}_i - \mathtt{SEP})$. The challenge is to find set SEP such that the value of (new) interference, which is $\lfloor \frac{I_i(D_i, \mathtt{HP}_i - \mathtt{SEP}, (m-m'))}{m-m'} \rfloor$, becomes as small as possible where SEP is the set of m' separated tasks selected from set $\mathtt{HP}_i$. In other words, the problem to solve is the following: **What is the best way to separate m' tasks from set $\mathtt{HP}_i$ such that the value of $I_i(D_i, \mathtt{HP}_i, m)$ is maximally reduced for some $m' > 0$?**

Note that when task τ_i fails to pass the DA-LC test before separation of any task from $\mathtt{HP}_i$, the value of $I_i(D_i, \mathtt{HP}_i, m)$ depends on $(m-1)$ carry-in tasks from set $\mathtt{HP}_i$. Let cis and ncs respectively denote the sets of CI tasks and NC tasks from set $\mathtt{HP}_i$ such that $\mathtt{HP}_i = (\mathtt{cis} \cup \mathtt{ncs})$. According to Eq. (6.7), $\mathtt{cis} = Max(\mathtt{HP}_i, m-1)$, and then obviously $\mathtt{ncs} = (\mathtt{HP}_i - \mathtt{cis})$. Separating each of the m' tasks from $\mathtt{HP}_i$ is equivalent to separating that task either from cis or ncs.

First, the criterion for separating exactly *one* task from $\mathtt{HP}_i$, particularly, separating one task either from set cis or ncs is considered. Then, based on this criterion of separating one task, the criteria for separating subsequent tasks is presented.

(Separation of one task) When $m' = 1$, either one CI-task or one NC-task is separated and this task is selected either from set cis or ncs, respectively. Remember that it is also needed to separate $m' = 1$ processor. Thus, the number of CI tasks after separation is at most $(m - m' - 1) = (m-2)$ when applying the DA-LC test to task τ_i considering the non-separated tasks from $\mathtt{HP}_i$ and $(m - m')$ processors.

When separating a CI-task τ_k, where $\tau_k \in \mathtt{cis} \subseteq \mathtt{HP}_i$, the value of $I_i(D_i, \mathtt{HP}_i, m)$ is reduced by $\mathtt{I}^{\mathtt{CI}}_{k,i}(D_i)$ (i.e., the carry-in interfering workload of task τ_i) according to Eq. (6.7). In order to maximally reduce the value of $I_i(D_i, \mathtt{HP}_i, m)$ by separating exactly one CI task from cis, the best criterion is to select the task from cis that has the *largest* value of carry-in interfering workload. The largest value of interfering carry-in workload of any task in cis is given as follows:

$$\max_{\tau_k \in \mathtt{cis}} \{\mathtt{I}^{\mathtt{CI}}_{k,i}(D_i)\}$$

Separating a NC-task τ_j, where $\tau_j \in \mathtt{ncs} \subseteq \mathtt{HP}_i$, has *two* effects. First, separating the NC task τ_j from ncs reduces the value of $I_i(D_i, \mathtt{HP}_i, m)$ by $\mathtt{I}^{\mathtt{NC}}_{j,i}(D_i)$ (i.e., the non interfering carry-in workload of τ_j). Second, one of the CI tasks from cis becomes a new NC task since, after separation, there are at most $(m-2)$ carry-in tasks. The CI task from cis that becomes a NC task is the one with the *minimum* value of the difference between its carry-in and non carry-in interfering workload among all the tasks in cis. This is because, after separation, the Max function in Eq. (6.7) would consider $(m-2)$ carry-in tasks that have the largest values of the difference between the carry-in and non carry-in interfering workload. Thus, separating a NC-task τ_j from ncs reduces the value of $I_i(D_i, \mathtt{HP}_i, m)$ by the following amount:

$$\mathtt{I}^{\mathtt{NC}}_{j,i}(D_i) + \underset{\tau_d \in \mathtt{cis}}{min} \{\mathtt{I}^{\mathtt{DIFF}}_{d,i}(D_i)\}$$

where $\underset{\tau_d \in \mathtt{cis}}{min} \{\mathtt{I}^{\mathtt{DIFF}}_{d,i}(D_i)\}$ is the *minimum* value of the difference between the carry-in and non carry-in interfering workload for any task in cis. Note that the value of $\underset{\tau_d \in \mathtt{cis}}{min} \{\mathtt{I}^{\mathtt{DIFF}}_{d,i}(D_i)\}$ is completely *independent* of the NC task τ_j that is selected for separation from ncs. Thus, in order to maximally reduce $I_i(D_i, \mathtt{HP}_i, m)$ by separating exactly one NC task from ncs, the best criterion is to select the NC task from ncs that has the *largest* value of non carry-in interfering workload. The largest value of non carry-in interfering workload of any task in ncs is given as follows:

$$\underset{\tau_j \in \mathtt{ncs}}{max} \{\mathtt{I}^{\mathtt{NC}}_{j,i}(D_i)\}$$

The criterion to determine whether to separate a CI task or a NC task, when $m' = 1$, is determined as follows.

Criterion For Separating One Task: When $m' = 1$, the task $\tau_a \in \mathtt{cis}$ that satisfies $\mathtt{I}^{\mathtt{CI}}_{a,i}(D_i) = \underset{\tau_k \in \mathtt{cis}}{max} \{\mathtt{I}^{\mathtt{CI}}_{k,i}(D_i)\}$ is selected for separation if

$$\underset{\tau_k \in \mathtt{cis}}{max} \{\mathtt{I}^{\mathtt{CI}}_{k,i}(D_i)\} > \left(\underset{\tau_j \in \mathtt{ncs}}{max} \{\mathtt{I}^{\mathtt{NC}}_{j,i}(D_i)\} + \underset{\tau_d \in \mathtt{cis}}{min} \{\mathtt{I}^{\mathtt{DIFF}}_{d,i}(D_i)\} \right) \tag{6.10}$$

otherwise, task $\tau_b \in \mathtt{ncs}$ satisfying $\mathtt{I}^{\mathtt{NC}}_{b,i}(D_i) = \underset{\tau_j \in \mathtt{ncs}}{max} \{\mathtt{I}^{\mathtt{NC}}_{j,i}(D_i)\}$ is selected for separation. □

(Separation of more than one task) If $m' > 1$, then one task from set $\mathtt{HP}_i = (\mathtt{cis} \cup \mathtt{ncs})$ is first separated using the criterion in Eq. (6.10). Then, this separated task, say task τ_s, is *removed* from either cis or ncs depending on whether $\tau_s \in \mathtt{cis}$ or $\tau_s \in \mathtt{ncs}$, respectively. Now separating the next task is the same as separating one new task from the updated set $(\mathtt{cis} \cup \mathtt{ncs}) = \mathtt{HP}_i - \{\tau_s\}$ using the criterion in Eq. (6.10). The pseudocode for selecting the m' tasks from set $\mathtt{HP}_i$ for separation is given in Figure 6.3. The algorithm Select(ψ, m', τ_i, t) in Figure 6.3 returns m' separated tasks selected from set ψ considering the target task τ_i and a problem window of size t.

The algorithm in Figure 6.3 has four parameters. The first parameter ψ is the set of higher priority tasks of the target task τ_i, the second parameter m' is the number of tasks that need to be separated from set ψ, the third parameter τ_i is the target task, and finally, the fourth parameter t is the length of the problem window. It will be evident later that the proposed priority assignment assignment policy for the IA-DA test separates m' higher priority tasks from set $\mathtt{HP}_i$ by calling Select$(\mathtt{HP}_i, m', \tau_i, D_i)$ before applying the DA-LC test for the target task τ_i considering the problem window of length D_i.

The set of CI tasks and NC tasks from set ψ are determined in line 1–2 of Figure 6.3 where set $Max(\psi, m'-1)$ is defined in Eq. (6.4). Each iteration of the loop in line 3–13 selects one task from $(\mathtt{cis} \cup \mathtt{ncs})$ for separation. The task to be separated during each

Algorithm Select(ψ, m', τ_i, t)

// ψ is the set of higher priority tasks of τ_i
// m' tasks needs to be separated from set ψ
// The target task is τ_i
// The problem window is of length t

1. $\texttt{cis} = Max(\psi, m' - 1)$
2. $\texttt{ncs} = \psi - \texttt{cis}$
3. For $g = 1$ to m' // each iteration separates one task
4. Find the task $\tau_a \in \texttt{cis}$ where $\mathtt{I}^{\mathtt{CI}}_{a,i}(t) = \max_{\tau_k \in \texttt{cis}} \mathtt{I}^{\mathtt{CI}}_{k,i}(t)$
5. Find the task $\tau_b \in \texttt{ncs}$ where $\mathtt{I}^{\mathtt{NC}}_{b,i}(t) = \max_{\tau_j \in \texttt{ncs}} \mathtt{I}^{\mathtt{NC}}_{j,i}(t)$
6. Find the task $\tau_c \in \texttt{cis}$ where $\mathtt{I}^{\mathtt{DIFF}}_{c,i}(t) = \min_{\tau_d \in \texttt{cis}} \mathtt{I}^{\mathtt{DIFF}}_{d,i}(t)$
7. If $(\mathtt{I}^{\mathtt{CI}}_{a,i}(t) > \mathtt{I}^{\mathtt{NC}}_{b,i}(t) + \mathtt{I}^{\mathtt{DIFF}}_{c,i}(t))$ Then
8. $\texttt{cis} = \texttt{cis} - \{\tau_a\}$
9. Else
10. $\texttt{cis} = \texttt{cis} - \{\tau_c\}$
11. $\texttt{ncs} = (\texttt{ncs} \cup \{\tau_c\}) - \{\tau_b\}$
12. End If
13. End For
14. Return $\psi - (\texttt{ncs} \cup \texttt{cis})$

Figure 6.3: *Algorithm for selecting the tasks for separation*

iteration is either a CI task from cis or a NC task from ncs. The CI task $\tau_a \in$ cis having the *largest* carry-in interfering workload is determined in line 4. The NC task $\tau_b \in$ ncs having the *largest* non carry-in interfering workload is determined in line 5. The CI task $\tau_c \in$ cis having the *smallest* value of the difference between its carry-in and non carry-in interfering workload is determined in line 6.

The condition in line 7 (based on the criterion in Eq. (6.10)) determines whether separation of the CI task τ_a or separation of the NC task τ_b would maximally reduce the value of $I_i(t, \psi, m)$. If the CI task τ_a is separated, i.e., condition in line 7 is true, then τ_a is removed from set cis in line 8. If the NC task τ_b is separated, i.e., condition in line 7 is false, then the CI task τ_c determined in line 6 becomes a NC task, and thus task τ_c is first removed from set cis in line 10. Then, task τ_c is included in set ncs, and finally, the NC task τ_b is removed from set ncs in line 11. Separation of the subsequent task in next iteration uses these updated sets of CI and NC tasks. When the for loop exits, the set of total m' separated tasks in $\psi - (\texttt{cis} \cup \texttt{ncs})$ is returned in line 14.

Lemma 6.1 now shows that the proposed separation criterion of algorithm Select in Figure 6.3 is *better* in terms of reducing the pessimism in interference estimation for the DA-LC test in comparison to that of the separation criterion that is based on the "highest-density" policy as proposed for the H-ODA-LC test.

Lemma 6.1. *If task τ_i passes the DA-LC test by separating m' highest-density tasks from set HP_i of higher priority tasks, then τ_i also passes the DA-LC test by separating the tasks returned by algorithm Select($\mathsf{HP}_i, m', \tau_i, D_i$) from set HP_i, where DA-LC test in both cases after separation uses $(m - m')$ processors and the non-separated tasks from set HP_i .*

Proof. Let $\mathsf{SEP}_{density}$ is the set of m' highest-density tasks from set HP_i and $H_{density} = (\mathsf{HP}_i - \mathsf{SEP}_{density})$. Let SEP_{new} is the set of m' tasks returned by the algorithm Select($\mathsf{HP}_i, m', \tau_i, D_i$) and $H_{new} = (\mathsf{HP}_i - \mathsf{SEP}_{new})$. If τ_i passes the DA-LC test by separating the tasks in $\mathsf{SEP}_{density}$ from HP_i, then according to the DA-LC test in Eq. (6.9), the following holds:

$$\left\lfloor \frac{I_i(D_i, H_{density}, m - m')}{m - m'} \right\rfloor \leq (D_k - C_k)$$

Note that the interfering workload of each task $\tau_k \in \mathsf{HP}_i$ for the DA-LC test is calculated based on static parameters of the task τ_k (i.e., independent of other tasks in HP_i). Algorithm Select at each stage separates from set HP_i the task that maximally reduces $I_i(D_i, \mathsf{HP}_i, m)$. Since the total interfering workload is the sum of interfering workload of the non-separated tasks, algorithm Select maximally reduces $I_i(D_i, \mathsf{HP}_i, m)$ by separating m' tasks from set HP_i, and we must have

$$I_i(D_i, H_{new}, m - m') \leq I_i(D_i, H_{density}, m - m')$$

Consequently, the following also holds

$$\left\lfloor \frac{I_i(D_i, H_{new}, m - m')}{m - m'} \right\rfloor \leq (D_k - C_k)$$

which implies that task τ_i also passes the DA-LC test. □

The two tasks (i.e., τ_3 and τ_4), separation of which makes the task set in Example 6.2 (page 94) schedulable, can be determined using the separation criterion of the Select algorithm presented in Figure 6.3; but can not be determined using the "highest-density" based separation criterion. Thus, the proposed separation criterion is *better* in terms of reducing the amount of pessimism in calculating the interference due to the higher priority tasks on a lower priority task. Now the details of the priority assignment policy for global FP scheduling based on this new separation criterion is presented. The IA-DA test, presented in Subsection 6.5.3, essentially combines the schedulability test and priority assignment of the tasks. And, successful priority assignment of all the tasks implies that the task set is schedulable using global FP scheduling.

6.5.3 Priority Assignment Algorithm: the IA-DA Test

The development of the priority assignment algorithm for the IA-DA test takes the advantage of the HPA policy based on the elegant separation criterion proposed in last subsection, applies the DA-LC test to each target task, and uses the basic idea of OPA algorithm for assigning the fixed priorities to the tasks. The priority assignment to the tasks in Γ starts from the lowest priority level $\mathtt{PL} = 1$ ends at the highest priority level $\mathtt{PL} = n$. The pseudocode of the priority assignment policy of the IA-DA test is presented in Figure 6.4.

Algorithm IA-DA(Γ , m)

1. $\Gamma_U = \Gamma$
2. For $\mathtt{PL} = 1$ to $(n - m)$
3. For each $\tau_i \in \Gamma_U$ //a new task is selected as target task
4. $\mathtt{HP}_i = \Gamma_U - \{\tau_i\}$
5. For $m' = 0$ to $(m - 1)$ //m' tasks from $\mathtt{HP}_i$ will be separated
6. $\mathtt{H} = \mathtt{HP}_i - \mathtt{Select}(\mathtt{HP}_i, m', \tau_i, D_i)$
7. If ($\left\lfloor \frac{I_i(D_i,H,m-m')}{m-m'} \right\rfloor + C_i \leq D_i$) Then
8. Task τ_i is assigned priority level PL
9. $\Gamma_U = \Gamma_U - \{\tau_i\}$
10. If (PL$= n - m$) Then
11. //there are m tasks left in Γ_U
12. Each task in Γ_U is assigned one unique
13. priority level between $(n - m + 1)$ to n
14. Return "Schedulable"
15. Else
16. Break and go to next priority level (line 2)
17. End If
18. End If
19. End For
20. End For
21. Return "Failure"
22. End For

Figure 6.4: *The IA-DA test*

Initially, all tasks are considered as potential target tasks for priority assignment at the lowest priority level PL=1. All the tasks in set Γ are stored in variable Γ_U (set of priority-unassigned tasks) in line 1. Each iteration of the loop in line 2–22 represents one priority level starting from the lowest priority level PL=1 to the highest priority level PL=(n-m). Note that priority assignment of the final m priority-unassigned tasks is trivial since these tasks are assigned the m highest priority levels. Therefore, the loop in line 2–22 runs from 1 to $(n - m)$ and tries to assign one priority-unassigned task one priority level.

At each priority level PL, the inner loop in line 3–20 considers one-by-one priority-unassigned task from set Γ_U until one such task is assigned the priority level PL. During each iteration of the loop in line 3–20, a new task $\tau_i \in \Gamma_U$ is selected as a target task in line 3. The set of other priority-unassigned tasks $\mathsf{HP}_i = (\Gamma_U - \{\tau_i\})$ is determined in line 4. If the target task τ_i is eventually assigned the priority level PL, then the tasks in set HP_i will have higher priorities than task τ_i.

For a given target task τ_i, the algorithm (temporarily) separates m' tasks from set HP_i and it also separates m' processors. During each iteration (using the iterative variable $m' = 0, \ldots (m-1)$) of the loop in line 5–19, a total of m' tasks from set HP_i are separated in line 6 by calling algorithm `Select(`$\mathsf{HP}_i, m', \tau_i, D_i$`)`. The other non-separated, priority-unassigned tasks from set HP_i are stored in set H in line 6 where $H = (\mathsf{HP}_i -$ `Select(`$\mathsf{HP}_i, m', \tau_i, D_i$`)`$)$. Notice that the set of separated tasks for each target task may be *different*. Next the DA-LC test is applied in line 7 to determine if the target task τ_i can be assigned priority level PL by assuming the higher priorities of the tasks in set H. In such case, the DA-LC test uses $(m - m')$ processors and only the higher priority tasks in set H.

If the DA-LC test in line 7 is satisfied, then task τ_i is assigned priority level PL in line 8 and removed from the set of priority-unassigned tasks in line 9. If the current priority level PL is equal to $(n - m)$, i.e., condition in line 10 is true, then there are exactly m (priority-unassigned) tasks in Γ_U after τ_i is removed from Γ_U in line 9. And, each of these m priority-unassigned tasks in Γ_U is assigned one unique priority level between PL=(n-m+1) and PL=n in line 12–13 (note that these are the m highest priority tasks and are always schedulable). At this point, all tasks are assigned priorities and the algorithm returns "schedulable" in line 14. If the current priority level PL is less than $(n - m)$, i.e., the condition in line 10 is false, then the priority assignment for next priority level starts (jumping from line 16 to line 2).

If the DA-LC test for task τ_i in line 7 is never satisfied for any m', $0 \leq m' < m$, then the for loop in line 5–19 exits; and the loop in line 3–20 selects another new target task. If no *new* task can be selected as a target task at line 3, then the for loop in line 3–20 exits. Since at this stage there is *no* task that is assigned the current priority level PL, the algorithm returns "Failure" in line 21.

Notice that if a target task can not be assigned priority level PL, the corresponding separated processors and separated tasks are *no* more considered "separated". And, these tasks along with other priority-unassigned tasks are considered as candidates for selecting the next target task at the current priority level. Similarly, if a target task is assigned priority level PL, the separated tasks along with other priority-unassigned tasks are considered as candidates for selecting the target tasks at next priority level. In other words, *the priority assignment for each new target task starts with all the priority-unassigned tasks, i.e., set Γ_U, and all the m processors*. It is not difficult to see that the time complexity of algorithm IA-DA is polynomial.

Correctness of the IA-DA Test: The correctness of the priority assignment policy of the IA-DA test is proved in Theorem 6.2 by showing that if the IA-DA test in Fig-

ure 6.4 successfully assigns the priorities, then all the deadlines are met. The following Lemma 6.2 will be used in Theorem 6.2.

Lemma 6.2. *Consider four positive integers w, x, y, and z. The following holds:*

$$\left\lfloor \frac{w}{x} \right\rfloor + y \leq z \text{ if and only if } w \leq x \cdot (z - y + 1) - 1$$

Proof. (**if part**) It will be shown that, if $w \leq x \cdot (z - y + 1) - 1$, then $\lfloor \frac{w}{x} \rfloor + y \leq z$. Since $w \leq x \cdot (z - y + 1) - 1$, then the following (due to integer assumption) is true

$$\begin{aligned} & w < x \cdot (z - y + 1) \equiv \frac{w}{x} < (z - y + 1) \\ \Rightarrow \; & (\text{since } \left\lfloor \frac{w}{x} \right\rfloor \leq \frac{w}{x}) \\ & \left\lfloor \frac{w}{x} \right\rfloor < (z - y + 1) \\ \Rightarrow \; & (\text{since } \left\lfloor \frac{w}{x} \right\rfloor \text{ and } (z - y + 1) \text{ are integers}) \\ & \left\lfloor \frac{w}{x} \right\rfloor \leq (z - y) \equiv \left\lfloor \frac{w}{x} \right\rfloor + y \leq z \end{aligned}$$

(**only if part**) It will be shown that, if $\lfloor \frac{w}{x} \rfloor + y \leq z$, then $w \leq x \cdot (z - y + 1) - 1$ holds. Since, $\lfloor \frac{w}{x} \rfloor + y \leq z$ and $(\frac{w}{x} - 1) < \lfloor \frac{w}{x} \rfloor$, the following is true

$$\begin{aligned} & (\frac{w}{x} - 1) + y < z \equiv \; w < x \cdot (z - y + 1) \\ \Rightarrow \; & (\text{since } x \cdot (z - y + 1) \text{ is an integer}) \\ & w \leq x \cdot (z - y + 1) - 1 \end{aligned}$$

□

Theorem 6.2. *If algorithm IA-DA in Figure 6.4 returns "schedulable", then all the tasks in set Γ meet deadlines using global* FP *scheduling on m processors according to the priorities assigned by IA-DA.*

Proof. If algorithm IA-DA in Figure 6.4 returns "schedulable", then each of the tasks in Γ is assigned a unique priority level between 1 to n. It will be proved that each task that is assigned a priority level using algorithm IA-DA meets all the deadlines.

If a task τ_i is assigned any priority level PL between $(n - m + 1)$ and n in line 12–13 of Figure 6.4, then task τ_i is one of the m highest-priority tasks. Since we have m processors, each task assigned any priority level between $(n - m + 1)$ and n meets all its deadlines. Now consider a task τ_i that is assigned priority level PL such that $1 \leq \mathsf{PL} < (n - m + 1)$. It will be shown that task τ_i meets all the deadlines.

Since $\mathtt{PL} < (n-m+1)$, task τ_i is assigned priority in line 8 of the IA-DA algorithm in Figure 6.4. This implies that the condition in line 7 is true and the following holds:

$$\left\lfloor \frac{I_i(D_i, H, m-m')}{m-m'} \right\rfloor + C_i \leq D_i \tag{6.11}$$

where $H = [\mathtt{HP}_i - \mathtt{Select}(\mathtt{HP}_i, m', \tau_i, D_i)]$ and the set $\mathtt{HP}_i$ (determined in line 4) is the set of all tasks having higher priorities than that of task τ_i.

Since Eq. (6.11) holds, the maximum interference that any job of task τ_i suffers due to the higher priority tasks in H is $\lfloor \frac{I_i(D_i,H,m-m')}{m-m'} \rfloor$. According to Lemma 6.2, Eq. (6.11) holds, if and only if,

$$I_i(D_i, H, m-m') \leq (m-m') \cdot (D_i - C_i + 1) - 1 \tag{6.12}$$

Therefore, *the upper bound on the total interfering workload due to the tasks in H within the problem window of any job of task τ_i is* $[(m-m') \cdot (D_i - C_i + 1) - 1]$.

Notice that after task τ_i is assigned priority level PL, the corresponding separated tasks (i.e., tasks in set $[\mathtt{HP}_i - H]$) are considered as target tasks at next higher priority levels and are ultimately assigned higher priority levels than task τ_i. Thus, task τ_i suffers interference not only from the tasks in set H but also from the "separated" tasks returned by the algorithm $\mathtt{Select}(\mathtt{HP}_i, m', \tau_i, D_i)$. The upper bound on interfering workload due to each of the tasks returned by the algorithm $\mathtt{Select}(\mathtt{HP}_i, m', \tau_i, D_i)$ is $(D_i - C_i + 1)$ according to Eq. (6.4) and Eq. (6.5) as given in page 87. Thus, the total interfering workload due to all the m' separated tasks, determined by calling $\mathtt{Select}(\mathtt{HP}_i, m', \tau_i, D_i)$, is at most $[m' \cdot (D_k - C_k + 1)]$. Thus, the total interfering workload due to all the higher priority tasks in $\mathtt{HP}_i = \mathtt{H} \cup \mathtt{Select}(\mathtt{HP}_i, m', \tau_i, D_i)$ on any job of task τ_i is at most:

$$\begin{aligned} &[(m-m') \cdot (D_k - C_k + 1) - 1] + [m' \cdot (D_k - C_k + 1)] \\ &= m \cdot (D_k - C_k) + (m-1) \end{aligned}$$

Because interference is an integer and all the m processors are simultaneously busy executing the tasks in $\mathtt{HP}_i$ when task τ_i is interfered, the interference that any job of task τ_i suffers (based on similar reasoning in [BC07, GSYY09, DB11b]) is at most $\lfloor \frac{m(D_k-C_k)+(m-1)}{m} \rfloor = (D_k - C_k)$. Consequently, any job of task τ_i meets its deadline. □

If IA-DA in Figure 6.4 returns "schedulable", then all the tasks in Γ meet deadlines using global FP scheduling on m processors according to the priorities assigned by IA-DA. The IA-DA test dominates the H-ODA-LC test as is given in next theorem.

Theorem 6.3. *If task set Γ is schedulable using the H-ODA-LC test, then Γ is also schedulable using the IA-DA test, and not conversely.*

Proof. Proof in given in Appendix A (page 224). □

6.6 The IA-RT Test

The algorithm for the IA-DA test in Figure 6.4 applies the DA-LC test in line 7 for target task τ_i by considering the higher priorities of the tasks in set H using $(m - m')$ processors and a problem window of length D_i. Remember that the response-time-based RTA-LC test dominates the deadline-analysis-based DA-LC test. However, the RTA-LC test can not be applied in line 7. This is because the response time R_k for each task $\tau_k \in H$ has to be known before applying the RTA-LC test for task τ_i in line 7. This way the RTA-LC test being OPA-incompatible can not be used in line 7 in Figure 6.4 for the IA-DA test. However, there is another response-time test proposed by Davis and Burns in [DB10], called the D-RTA-LC test, which uses the same schedulability analysis as the DA-LC test but uses a problem window that is never larger than that of considered for the DA-LC test. The D-RTA-LC test is OPA-compatible and dominates the DA-LC test. Based on these observations, the IA-DA test is further improved by using the D-RTA-LC test and the proposed criterion for separating the tasks when determining the schedulability and priority of each target task τ_i.

In this chapter, the IA-DA test is improved by incorporating the D-RTA-LC test rather than using the DA-LC test to determine whether a target task can be assigned a particular priority level. First, the D-RTA-LC test is presented in subsection 6.6.1. Then, the IA-RT test and its priority assignment policy are proposed in subsection 6.6.2.

6.6.1 The D-RTA-LC Test

The D-RTA-LC test [DB10] is similar to the RTA-LC test except that it uses the CI workload computation of the DA-LC test (given in Eq. (6.3)) instead that of the RTA-LC test (given in Eq. (6.2)). The details of the D-RTA-LC test are given below:

Workload. The NC workload $\mathtt{W}_k^{\mathtt{NC}}(t)$ of τ_k in an interval of length t is given as follows:

$$\mathtt{W}_k^{\mathtt{NC}}(t) = \lfloor t/T_k \rfloor \cdot C_k + min(C_k, t - \lfloor t/T_k \rfloor \cdot T_k) \tag{6.13}$$

The CI workload $\mathtt{W}_k^{\mathtt{CI}}(t)$ of τ_k in an interval of length t is given as follows:

$$\mathtt{W}_k^{\mathtt{CI}}(t) = A_t^k \cdot C_k + min(C_k, t + D_k - C_k - A_t^k \cdot T_k) \tag{6.14}$$

where $A_t^k = \lfloor (t + D_k - C_k)/T_k \rfloor$.

Interfering Workload: The CI and NC interfering workload $\mathtt{I}_{k,i}^{\mathtt{CI}}(t)$ and $\mathtt{I}_{k,i}^{\mathtt{NC}}(t)$ are given as follows:

$$\mathtt{I}_{k,i}^{\mathtt{CI}}(t) = min(\mathtt{W}_k^{\mathtt{CI}}(t), t - C_i + 1) \tag{6.15}$$

$$\mathtt{I}_{k,i}^{\mathtt{NC}}(t) = min(\mathtt{W}_k^{\mathtt{NC}}(t), t - C_i + 1) \tag{6.16}$$

The difference between the CI and NC interfering workload of task τ_k within the problem window of length t is denoted by $\mathtt{I}^{\mathtt{DIFF}}_{k,i}(t)$ such that:

$$\mathtt{I}^{\mathtt{DIFF}}_{k,i}(t) \ = \mathtt{I}^{\mathtt{CI}}_{k,i}(t) \ - \mathtt{I}^{\mathtt{NC}}_{k,i}(t) \tag{6.17}$$

Total Interfering Workload. The upper bound on total interfering workload due to all the tasks in set $\psi \subseteq \mathtt{HP}_i$ is denoted by $\mathtt{I}_i(t, \psi, m)$. The value of $\mathtt{I}_i(t, \psi, m)$ is calculated as follows:

$$\mathtt{I}_i(t, \psi, m) = \sum_{\tau_k \in \psi} \mathtt{I}^{\mathtt{NC}}_{k,i}(t) + \sum_{\tau_k \in Max(\psi, m-1)} \mathtt{I}^{\mathtt{DIFF}}_{k,i}(t) \tag{6.18}$$

where $Max(\psi, m-1)$ is the set of $(m-1)$ tasks from set ψ that have the largest values of $\mathtt{I}^{\mathtt{DIFF}}_{k,i}(t)$.

Interference. An upper bound on interference due to the tasks in ψ on any job of task τ_i within the problem window of length t is $\lfloor \mathtt{I}_i(t, \psi, m)/m \rfloor$.

The D-RTA-LC Test: The D-RTA-LC test [DB10], which involves computing the upper bound on the response time of each task $\tau_k \in \Gamma$, is recursively given as follows for finding the response time $\overline{R}_i$ of task τ_i:

$$\overline{R}_i^{(h+1)} \leftarrow C_i + \left\lfloor \frac{\mathtt{I}_i(\overline{R}_i^h, \mathtt{HP}_i, m)}{m} \right\rfloor \tag{6.19}$$

Note that, in contrast to the RTA-LC test that computes R_i using Eq. (6.8), the response-time $\overline{R}_i$ of task τ_i based on Eq. (6.19) does not need to know the response time of the higher priority tasks $\tau_k \in \mathtt{HP}_i$. It is not difficult to see that the three conditions for OPA-compatibility (page 83) are satisfied for the D-RTA-LC test.

6.6.2 Priority Assignment Algorithm: the IA-RT Test

The IA-RT test is presented in Figure 6.5. The algorithm IA-RT in Figure 6.5 has two parameters: the task set Γ and the number of processors m. It determines whether the task set is schedulable on m processors by finding appropriate priority ordering of the tasks in Γ. The algorithm IA-RT in Figure 6.5 is similar to the algorithm IA-DA in Figure 6.4 with two major difference: (i) the OPA-compatible response time test D-RTA-LC in Eq. (6.19) is used to determine whether a target task can be assigned certain priority level, and (ii) the m' separated tasks are redetermined each time the size of the problem window changes.

Initially, all tasks are considered as potential target tasks for priority assignment at the lowest priority level PL=1. All the tasks in set Γ are stored in variable Γ_U (set of priority-unassigned tasks) in line 1. Each iteration of the loop in line 2–29 represents one priority level starting from the lowest priority level PL=1 to the highest priority level PL=(n-m).

At each priority level PL, the loop in line 3–27 considers priority-unassigned task from Γ_U until one such task is assigned the priority level PL. During each iteration of

Algorithm IA-RT(Γ , m)

1. $\Gamma_U = \Gamma$
2. For $\mathsf{PL} = 1$ to $(n - m)$
3. For each $\tau_i \in \Gamma_U$
4. $\mathsf{HP}_i = \Gamma_U - \{\tau_i\}$
5. For $m' = 0$ to $(m - 1)$
6. $\overline{R}_i^0 = C_i$
7. For $\mathsf{h} = 0$ to ∞
8. $\mathsf{H} = \mathsf{HP}_i - \mathtt{Select}(\mathsf{HP}_i\ , m', \tau_i, \overline{R}_i^h)$
9. $\overline{R}_i^{(h+1)} \leftarrow C_i + \left\lfloor \frac{\mathtt{I}_i(\overline{R}_i^h, H, m-m')}{m-m'} \right\rfloor$
10. **If** $\overline{R}_i^{(h+1)} = \overline{R}_i^h$ **Then**
11. Task τ_i is assigned priority level PL
12. $\Gamma_U = \Gamma_U - \{\tau_i\}$
13. If ($\mathsf{PL} = n - m$) Then
14.
15. Each task in Γ_U is assigned one unique
16. priority level between $(n - m + 1)$ to n
17. **Return** "Schedulable"
18. **Else**
19. Break and Go to next priority level (line 2)
20. **End If**
21. **End If**
22. **If** $\overline{R}_i^{(h+1)} > D_i$**Then**
23. Break and go to next iteration in line 5
24. **End If**
25. **End For** // loop with variable h in line 7 ends
26. **End For** // loop with variable m' in line 5 ends
27. **End For** // loop with variable τ_i in line 3 ends
28. **Return** "Failure"
29. **End For** // loop with variable PL in line 2 ends

Figure 6.5: *The IA-RT test*

the loop in line 3–27, a new task $\tau_i \in \Gamma_U$ is selected as a target task in line 3. The set of other priority-unassigned tasks $\mathsf{HP}_i = (\Gamma_U - \{\tau_i\})$ is determined in line 4. If the target task τ_i is eventually assigned the priority level PL, then the tasks in HP_i will have higher priorities than task τ_i.

For a given target task τ_i, the algorithm (temporarily) separate m' tasks from set HP_i considering the length of the current problem window and it also separates m' processors. During each iteration (using the iterative variable $m' = 0, \ldots (m-1)$) of the

loop in line 5–26, the response time of task τ_i is calculated based on the D-RTA-LC test in Eq. (6.19) by separating total of m' tasks from set HP_i in line 8.

The initial value of the problem window $\overline{R}_i^0$ is set to C_i in line 6. Remember that in response-time-based analysis if the response time of task τ_i is greater than the length of the current problem window, the size of the problem window is increased until the problem window is not greater than the relative deadline of the task. The for loop in line 7–25 determines the response time of task τ_i for each possible size of the problem window. A total of m' tasks is separated from set HP_i by considering the current problem window of size $\overline{R}_i^h$ (i.e., for the current value of the loop variable h) by calling the algorithm Select($\mathsf{HP}_i, m', \tau_i, \overline{R}_i^h$). The other non-separated, priority-unassigned tasks are stored in set H in line 8 where $H = (\mathsf{HP}_i - \texttt{Select}(\mathsf{HP}_i, m', \tau_i, \overline{R}_i^h))$. The size of the new problem window $\overline{R}_i^{(h+1)}$ is calculated in line 9 based on Eq. (6.19). If the length of the new problem window size has not increased (i.e., the response time calculation converges), then the target task τ_i can be assigned priority level PL.

If the D-RTA-LC test in line 10 is satisfied, then task τ_i is assigned priority level PL in line 11 and removed from the set of priority-unassigned tasks in line 12. If the current priority level PL is equal to $(n - m)$, i.e., condition in line 13 is true, then there are exactly m (priority-unassigned) tasks in Γ_U after τ_i is removed from Γ_U in line 12. And, each of these m priority-unassigned tasks in Γ_U is assigned one unique priority level between PL=(n-m+1) and PL=n in line 15–16 (note that these are the m highest priority tasks and are always schedulable). At this point, all tasks are assigned priorities and the algorithm returns "schedulable" in line 17. If the current priority level PL is less than $(n - m)$, i.e., the condition in line 13 is false, then the priority assignment for next priority level starts (jumping from line 19 to line 2).

If the D-RTA-LC test for task τ_i in line 10 is not satisfied for current m' and the new problem window size $\overline{R}_i^{(h+1)} > D_i$ in line 22, then separating one more tasks is considered by jumping from line 23 to line 5 in next iteration. If the D-RTA-LC test for task τ_i in line 10 is never satisfied for any m', $0 \leq m' < m$, then the for loop in line 5–26 exits, and the next iteration of loop in line 3 begins by selecting another new target task. If no new task can be selected as a target task at line 3, then the for loop in line 3–27 exits. Since at this stage there is *no* task that is assigned the current priority level PL, the algorithm returns "Failure" in line 28.

The correctness of the IA-RT test follows from the correctness of the IA-DA test proved in Theorem 6.2. Moreover, the IA-RT test dominates the IA-DA test since $\mathtt{I}_i(\overline{R}_i^h, H, m - m')$ in line 10 is never greater than $\mathtt{I}_i(D_i, H, m - m')$ as is used for the IA-DA test. In next section, the simulation results to compare the three proposed tests (H-ODA-LC, IA-DA, and IA-RT) with the state-of-the-art ODA-LC test are presented.

6.7 Empirical Investigation

In this section, empirical investigation into the performance of the proposed schedulability tests of global FP scheduling is presented. The derivation of theoretical result, for example, dominance of one schedulability test over another, does not demonstrate the average improvement of one test over another. Experimental investigation of an iterative schedulability test is highly effective in comparing different schedulability tests using randomly generated task sets. The `ODA-LC` test proposed by Davis and Burns [DB11b] is the sate-of-the-art iterative global FP schedulability test for constrained-deadline tasks. Each of the three tests (i.e., `H-ODA-LC`, `IA-DA`, and `IA-RT`) proposed in this chapter dominates the `ODA-LC` test. To quantitatively measure the improvement of the proposed tests over the state-of-the-art `ODA-LC` test, simulation using randomly generated task sets are conducted. The empirical investigation into the following four schedulability tests in Table 6.1 are presented in this section.

`ODA-LC` Test	The OPA algorithm in Figure 6.1 combined with the `DA-LC` test (proposed by Davis and Burns [DB11b]).
`H-ODA-LC` Test	The algorithm in Figure 6.2 (proposed in this thesis, page 91).
`IA-DA` Test	The algorithm in Figure 6.4 (proposed in this thesis, page 99).
`IA-RT` Test	The algorithm in Figure 6.5 (proposed in this thesis, page 105).

Table 6.1: *Different Iterative Schedulability Tests*

The metric, called *acceptance ratio*, is used to evaluate the effectiveness of each schedulability test. The acceptance ratio of a schedulability test is the percentage of the randomly generated task sets that are deemed schedulable using that schedulability test at a given utilization level. The larger the value of acceptance ratio at a utilization level, the better is the test in determining the global FP schedulability of task sets at that utilization level.

The `UUnifast-Discard` algorithm presented in subsection 5.6.1 (page 66) is used to generate n utilization values of a task set with cardinality n and total utilization U. Once a set of n utilizations $\{u_1, u_2, \ldots u_n\}$ of a task set is generated, the other parameters of each task τ_i are generated as follows:

- The minimum inter-arrival time T_i of each task τ_i is generated from the uniform random distribution within the range $[10ms, 1000ms]$.
- The WCET of task τ_i is set to $C_i = u_i \cdot T_i$.
- The relative deadline D_i of task τ_i is generated from the uniform random distribution within the range $[C_i, T_i]$.

Each of the experiments is characterized by a pair (m, n) where m is the number of processors and n is the cardinality of task set. For each experiment (m, n), task sets are generated at 40 different utilization levels: $\{0.025m, 0.5m, \ldots 0.975m, m\}$. A total of 1000 task sets at each of the 40 utilization levels using the `UUnifast-Discard` algorithm with parameters n and U (where U is the utilization level) are generated. Each of the 1000 task sets generated at a particular utilization level, say U, has cardinality n and total utilization equal to U. The schedulability of each of the 1000 task sets generated at each utilization level are determined based on the schedulability test for each of the four priority assignment policies in Table 6.1 and the acceptance ratio for each test is computed.

6.7.1 Result Analysis

A series of experiments for different pairs of (m, n) where $m \in \{2, 4, 8, 16\}$ and $n \in \{10, 20, 40, 60, 80, 160\}$ for constrained-deadline tasks are conducted. The acceptance ratios at each of the 40 utilization levels for each of the fours tests in Table 6.1 are calculated for each experiment. The important trends and observations based on these experiments are presented in this section.

In each graphs presented in this section, the x-axis represents the system utilization U/m for utilization level U and the y-axis represents the acceptance ratio. The acceptance ratios of all tests are around 100% at relatively low utilization level (e.g., $U \leq 0.3m$) and 0% at very high utilization level (e.g., $U > 0.85m$). The acceptance ratios for system utilization between 30% to 85%, which correspond to the utilization levels between $0.3m$ and $0.85m$, are plotted.

The impact of task set cardinality on the theoretically best `IA-RT` schedulability test is first discussed based on experimental results. It will be evident that when the cardinality of the task set is $\approx 5m$, then the acceptance ratio of the `IA-RT` test becomes relatively small, and it is concluded that $n = 5m$ represents the worst-case parameter setting regarding the task set generation algorithm for the proposed schedulability tests. Then, the comparison among all the four schedulability tests in Table 6.1 is presented to see the improvement of the proposed tests over the state-of-the-art `ODA-LC` test for task set cardinality equal to $5m$.

Impact of n on the `IA-RT` Test

In order to measure the impact of task set cardinality in determining the schedulability of random task sets using the `IA-RT` test for some given m, the acceptance ratios for experiments with $(m = 4, n)$ where $n = 8, 10, 12, 15, 20$ are presented in Figure 6.6. The acceptance ratios of the `IA-RT` test at each utilization level decreases as the task set size increases from 8 to 20 for a given m. It seems to be more difficult to schedule task sets with larger cardinality. This reason can be explained as follows: as the cardinality of the task set increases, the number of tasks having relatively large utilization also increases. Each of such heavy utilization tasks in the worst-case may occupy one

processor and leaving relative fewer number of free processors for other tasks. Consequently, the other tasks can not be decided to be schedulable using the IA-RT test on an insufficient number of processors.

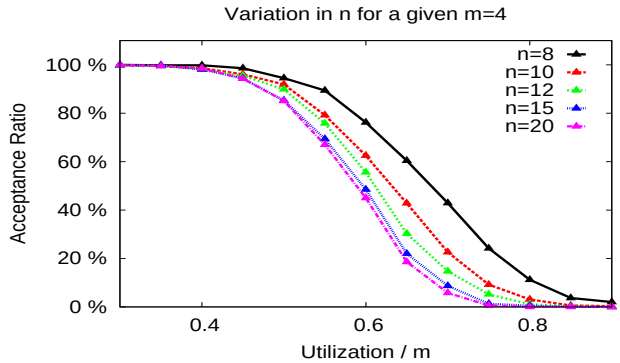

Figure 6.6: *Acceptance ratios of the IA-RT test for experiments with* $m = 4$ *and* $n = 8, 10, 12, 15, 20$.

When the cardinality is increased from 8 to 20 for $m = 4$, the decrease in acceptance ratios of the IA-RT test, due to having relatively higher number of large utilization tasks, only tells one-side of the story. If the cardinality of the task sets is increased beyond a certain number (e.g., $n \geq 5m$), then the trend is reversed: acceptance ratio at each utilization level increases with the increase in number of tasks in a task set. The acceptance ratios for experiments with $(m = 4, n)$ where $n = 20, 40, 60, 80, 100$ for the IA-RT test are presented in Figure 6.7.

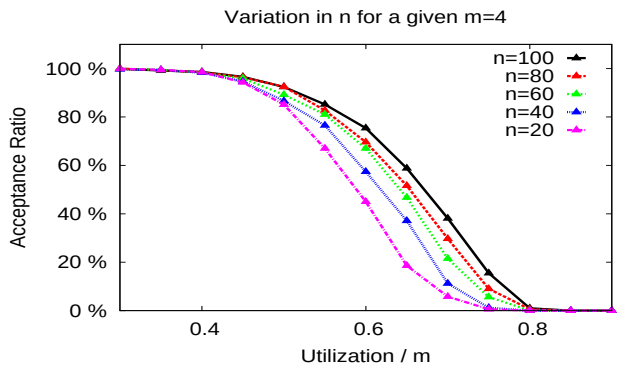

Figure 6.7: *Acceptance ratios of the IA-RT test for experiments with* $m = 4$ *and* $n = 20, 40, 60, 80, 100$.

In such case, the acceptance ratios of the IA-RT test increases as the task set size increases for a given m. This phenomenon can be explained as follows: as the cardinality

of each task set increases beyond $5m$, the number of high utilization tasks starts decreasing since the total utilization of the task set is now distributed across higher number of tasks. A low utilization task uses less computing resource and provides more opportunity for other tasks to execute on the processors. And, task set with smaller number of high utilization tasks does not suffer much from Dhall's effect.

The conclusion from these experiments is that $(m, 5m)$ seems to be the worst-case parameters for the experimental setup. To compare the improvement of the proposed tests in comparison to the state-of-the-art ODA-LC test, results related to the experimental parameter $n = 5m$ are only presented in this section. The experiments with $m = 4, 8, 16$ and $n = 3m, 10m$ are given in the Appendix B.

Observation 1: Remember that the average total density of task set increases as the cardinality of task set increases for a fixed number of processors (please see Figure 5.10 and Figure 5.11 in Chapter 5). While the acceptance ratio of the density-based tests proposed in Chapter 5 decreases with the increase in task set cardinality for a fixed number of processors, the iterative test IA-DA shows a different trend: the acceptance ratio decreases until $n = 5m$ and then increases again. This demonstrates that iterative tests are highly effective for scheduling tasks with large total density where the cardinality of a task set is relatively large.

Experiments with $(m, 5m)$

The acceptance ratios of all the four tests in Table 6.1 with experimental parameters $(m, 5m)$ are presented in Figure 6.8—6.10 where $m = 4, 8, 16$ are considered.

Observation 2: The acceptance ratios for the IA-DA and IA-RT tests do not differ noticeably although IA-RT test theoretically dominates the IA-DA test. The two plots for the IA-DA and IA-RT tests in Figure 6.8—6.10 are completely overlapping (i.e., difficult to see them separately). By looking at the raw acceptance ratio numbers of these two tests, it is found that those values are the same for almost all utilization levels and differ very insignificantly in the remaining utilization levels. The IA-DA test runs in polynomial time while the IA-RT test runs in pseudo-polynomial time. Given the polynomial time complexity of the IA-DA test and the fact that its performance is equivalent to the IA-RT test, the IA-DA test is the preferable iterative schedulability test. The discussion regarding the IA-DA test is thus also valid for the IA-RT test.

Observation 3: The improvement of the proposed three tests in this chapter over the state-of-the-art ODA-LC test is noticeable at higher utilization levels. The improvement in acceptance ratio of the proposed tests at higher utilization levels is due to improved priority assignment policy based on the HPA policy. By prudently separating the problematic tasks from the schedulability analysis of a target task, significant fraction of the randomly generated task sets pass one or more of the proposed tests but do not pass the ODA-LC test. The proposed IA-DA tests performs much better than the proposed H-ODA-LC test. This demonstrates the effectiveness of the novel separation criteria proposed for the IA-DA test in comparison to the highest-density based separation criteria proposed for the H-ODA-LC test.

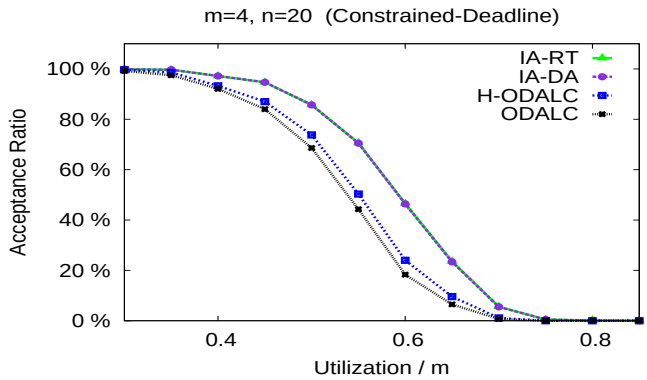

Figure 6.8: *Acceptance ratios for experiments with* $(m = 4, n = 5m = 20)$.

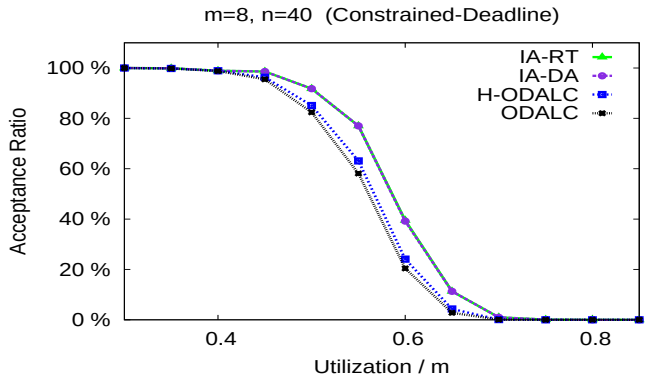

Figure 6.9: *Acceptance ratios for experiments with* $(m = 8, n = 5m = 40)$.

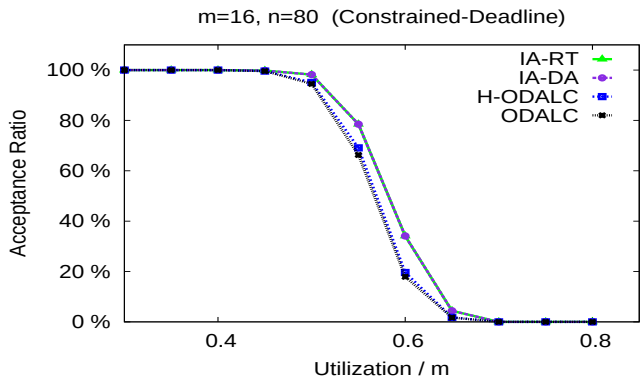

Figure 6.10: *Acceptance ratios for experiments with* $(m = 16, n = 5m = 80)$.

The IA-DA test significantly outperforms the state-of-the-art ODA-LC test. For example, the acceptance ratio of the IA-DA test at $0.6m$ utilization level for $(m = 8, n = 40)$ in Figure 6.9 is 38.5% while that of for the ODA-LC test is 16.4% (i.e., an improvement in acceptance ratio of more than 134%). Similarly, the acceptance ratio of the IA-DA test at $0.6m$ utilization level for $(m = 4, n = 20)$ in Figure 6.8 is approximately 47.3% while that of for the ODA-LC test is approximately 19.3% (an improvement in acceptance ratio of more than 145%).

Observation 4: The differences in acceptance ratio between the ODA-LC test and each of the other three proposed tests decreases as the number of processors increases. And, the acceptance ratios at each utilization level decreases for each of the four tests as the number of processors increases. For example, the plots of the acceptance ratios of IA-DA test in Figure 6.8—6.10 are becoming relatively "healthier" with decreasing m. This is due the way the task sets are generated for the experiments. When the number of processors is large, the number of tasks in a task set for experiments with $(m, 5m)$ is also relatively larger (one additional processor causes the cardinality to increase by 5).

Given that the number tasks in a task set is larger, the interference on the problem window of each target task is still too large even after separating at most $0, 1, \ldots (m - 1)$ problematic tasks. There are too many problematic tasks such that separation can not sufficiently reduce interference. And, each lower priority task suffers interference from a relatively larger number of higher priority tasks each of which contributes to the computation of interference. The interference is possibly relatively higher on a lower priority task for task sets with larger cardinality. Therefore, the acceptance ratio of all the tests decreases at each utilization level with increasing m.

6.8 Summary

This chapter proposes three different iterative schedulability tests for global FP scheduling: H-ODA-LC test, IA-DA test, and IA-RT test. Each of these proposed tests dominates the state-of-the-art ODA-LC schedulability test. All these proposed tests is based on HPA policy which is effective in reducing the amount of pessimism in the calculation of interference when analyzing the schedulability of a particular task. It has been shown that separating the highest-density tasks, as is done for the proposed H-ODA-LC test, is not the best choice of separated tasks for the HPA policy. A novel strategy to find the best set of separated tasks when considering the schedulability analysis of a lower priority task is proposed for the IA-DA and the IA-DA tests.

Both the proposed IA-DA and IA-RT tests perform significantly better than the state-of-the-art ODA-LC test. While the time complexity in evaluating the IA-DA test is polynomial, the time complexity in evaluating the IA-RT test is pseudo-polynomial. Although the IA-RT test dominates the IA-DA test, empirical investigation shows that the performance difference between these two tests is insignificant. This finding implies that one should apply the polynomial-time IA-DA test first before applying the pseudo-polynomial IA-RT test to determine the FP schedulability of a task set.

7

Fault-Tolerant Scheduling on Uniprocessor

A fault-tolerant deadline-monotonic (FTDM) scheduling of constrained-deadline sporadic tasks for tolerating multiple task errors on uniprocessor is presented in this chapter. Time-redundant execution of backup tasks is considered to recover from task errors. Each task has multiple backups that are scheduled one-by-one until the output of the task is correct. The fault model that FTDM scheduling considers is very powerful in the sense that it includes multiple hardware or software faults that can cause errors at any time, in any task, and even during the recovery. Tolerating a task error by executing its backup means that the task is able to produce its correct output before the deadline.

The schedulability analysis of the FTDM scheduling is based on computing the workload of each task and its higher priority tasks within an interval equal to the relative deadline of the task under study. The schedulability analysis of the FTDM scheduling derives an exact test considering at most f task errors within each of all possible intervals of length equal to the maximum relative deadline of any task.

7.1 Introduction

The importance of dependability on computer systems is increasing as computers are taking a more active role in everyday control applications. Fault-tolerance in such systems is an important aspect to guarantee the correctness of the application even in the event of faults. In many safety-critical systems, use of time redundancy is considered as a cost-efficient means to achieve fault-tolerance. In such systems, when a task error

is detected, the backup of the task is executed. However, due to the additional real-time requirements, it is essential that exploitation of time as a means for tolerating faults must not compromise the timeliness guarantee in the system.

The two requirements, achieving fault-tolerance through time redundancy and meeting the deadlines of the tasks, seem to be antagonistic. To guarantee both the correctness and timeliness of dependable real-time systems, it is necessary to design fault-tolerant scheduling algorithm and to derive appropriate schedulability test. An algorithm, called Fault-Tolerant Deadline-Monotonic (FTDM) scheduling, is proposed and its schedulability analysis is presented in this chapter. The proposed FTDM scheduling algorithm is based on FP scheduling on uniprocessor where the tasks are given the Deadline-Monotonic (DM) priorities. However, the FTDM scheduling and its schedulability analysis are also applicable to arbitrary fixed-priority assignment of the tasks.

The fault model (presented in Section 3.3) of the FTDM algorithm considers the occurrences of at most f task errors within each of the all possible intervals of length D_{max} where D_{max} is the largest relative deadline of any task in the sporadic task set Γ. There is no assumption regarding the distribution of the faults or on minimum inter-arrival time of the faults that could cause task errors. Relaxing these assumptions allow to consider many different situations, for example, where (i) a single job of a particular task is affected by multiple faults, (ii) different jobs of different tasks might be affected by multiple faults, (iii) faults that may occur in bursts, and (iv) the inter-arrival time of consecutive faults is not predictable.

The FTDM scheduling considers passive backups: no backup is dispatched until a task error is detected. Each task is considered to have one primary and several backups, where a backup could be same as the primary or could be a diverse implementation of the same task. The worst-case execution time of the backups associated with a particular task may be different. The backups associated with a particular task have the same priority as the primary and these backups are scheduled by FTDM algorithm one-by-one until the no task error is detected. The time-redundant execution of backups to recover from task errors takes additional CPU time. The FTDM algorithm requires to ensure that the correct output of each job of each task is generated before its deadline even if execution of backups are required to tolerate task errors.

The objective of the schedulability analysis of the FTDM algorithm is to derive a schedulability test that needs to be verified to ensure that all the deadlines are met. The outcome of the schedulability analysis of FTDM algorithm is the derivation of an *exact* schedulability test. The exact test is derived for each task (an iterative test) and based on computing the maximum total workload requested within the release time and deadline of any job of each task. To calculate the maximum total workload considering occurrences of task errors, a novel technique to *compose* the execution time of the higher priority jobs is used.

The only work that deals with a similar fault model as the FTDM algorithm is addressed by Aydin [Ayd07], but considered EDF priority and the exact test in [Ayd07] has an exponential run-time complexity. On the other hand, the run time-complexity to evaluate the exact schedulability test of the proposed FTDM algorithm is $O(n \cdot \hat{N} \cdot f^2)$,

where $\hat{N}$ is the maximum number of jobs (generated by the n periodic tasks) released within any time interval of length D_{max}. No previous work has derived an exact fault-tolerant uniprocessor schedulability test that has a lower time complexity than that is presented in this thesis for the assumed fault model.

The FTDM algorithm does not consider tolerating processor failures. Fault-tolerant multiprocessor scheduling algorithm for tolerating both task errors and processor failures is proposed in Chapter 8. However, the uniprocessor schedulability analysis of FTDM algorithm is applicable to partitioned multiprocessor scheduling in which each processor executes (preassigned) tasks based on uniprocessor FP scheduling algorithm. The exact uniprocessor schedulability condition of the FTDM algorithm can be applied during task-to-processor assignment phase in partitioned multiprocessor scheduling. To determine whether an unassigned task can be feasibly assigned to a processor, the proposed exact test for FTDM scheduling can be used to guarantee that each processor can tolerate up to f task errors within any time interval equal to the maximum relative deadline of the tasks assigned to that particular processor.

The rest of the chapter is organized as follows: the system model and the FTDM algorithm are presented in Section 7.2. Then, the related work on fault-tolerant scheduling on uniprocessor is presented in Section 7.3. The problem statement is formally given in Section 7.4. The schedulability analysis of one lower priority task under the FTDM scheduling is presented in Section 7.5. Then, in Section 7.6, the exact test of the entire task set is derived. The pseudocode of the exact test for FTDM scheduling is presented in Section 7.7 and its applicability to the multiprocessor setting is discussed. Section 7.8 summarizes this chapter.

7.2 System Model

The task and fault models for FTDM scheduling are presented in Section 3.1 and Section 3.3, respectively. The salient features of the models are reiterated here for readability. A set of n constrained-deadline sporadic tasks $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ is considered where each task $\tau_i \in \Gamma$ is characterized by WCET C_i, relative deadline D_i, and period T_i. At most f task errors due to a variety of hardware and software faults may occur within each of the all possible time intervals of length D_{max}. The f task errors may occur in the same job or may occur in different jobs of different tasks. The WCET of the primary of task τ_i is C_i and the WCET of each of the f backups of task τ_i is denoted by E_i^k where $k = 1, 2, \ldots f$.

Scheduler Model. The FTDM scheduling is uniprocessor FP scheduling where each task's primary or backup is executed based on DM priority ordering. And, each backup of task τ_i has the same priority as that of task τ_i. The FTDM scheduling works as follows. For each task τ_i, whenever a job of tis task is released, the primary executes first. If an error is detected at the end of execution of the primary, the first backup of the task becomes ready for execution. Again an error may be detected at the end of execution of this backup which in turn would trigger the execution of next backup, and so on. Each

task is considered to have f different backups in case all the f task errors occur in the same job of the task.

Remember that during the execution of a particular primary or backup of a task, at most one fault could affect this execution; and each error is assumed to be detected at the end of execution of a primary or backup (please see the fault-model in Section 3.3). It is assumed that there is no fault propagation: one error can affect exactly one primary or backup. If the cumulative execution demand within an interval of length D_{max} due to f task errors is maximum, then it is necessary that all the f task errors occur within that interval. If total k task errors, $k \leq f$, affect a particular job of task τ_i, then the execution time required for recovery is *maximized* if the first error affect the primary and each of the subsequent $(k-1)$ errors affect each subsequent backup of the same job of task τ_i.

The exact FTDM schedulability condition has to check that whether all the tasks deadlines are met or not if the occurrences of task errors is not worse than the assumed fault model. Since there are many different combinations of the occurrence of task errors that could affect the execution of the tasks in an interval of length D_{max}, algorithm FTDM must guarantee that the schedule is fault-tolerant for each such combination. In other words, all tasks must met their deadlines for any combination of errors affecting the different jobs of different tasks. The different combination of errors lead to the notion of *fault pattern*.

Fault-Pattern. Remember that there are a maximum of $\hat{N}$ jobs released within any interval of length D_{max}. There are different possibilities of the occurrences of the f errors affecting the $\hat{N}$ jobs. One possibility is that all the f errors occur in one of the $\hat{N}$ jobs. Another possibility is that different number of errors occur in different jobs. Each such possibility of error occurrence is called a *fault pattern* in [Ayd07, LMM00]. Given the jobs in set A, any possible combination of k errors that can affect the jobs in set A is denoted by k-fault-pattern. For example, if $k = 0$, no error occurs within the jobs in set A. If $k = 1$ and $|A| = 5$, then there are 5 different 1-fault-patterns since the single error due to the fault may affect any one of the five jobs in set A.

To achieve fault-tolerance, it has to be ensured that all the jobs released in any interval of length D_{max} meet the deadlines for f-fault patterns. The question that arises is: *what are the different possible fault patterns that one must consider for FTDM schedulability analysis of $\hat{N}$ jobs released within a time interval of length D_{max}?* In other words, in how many ways the f task errors could affect the $\hat{N}$ jobs that are released within any time interval of length D_{max}. It is already pointed out in [Ayd07] that the number of different fault patterns is given by the binomial coefficient $\binom{\hat{N}+f-1}{f} = \Omega((\frac{\hat{N}}{f})^f) = O(\hat{N}^f)$, which is exponential [CLRS01]. The FTDM schedulability analysis on uniprocessor considering this exponential number of different fault patterns may not be computationally practical if f are large. To overcome this problem, a dynamic-programing technique is used to find an exact FTDM schedulability condition. The time complexity of this technique for evaluating the exact test is $O(n \cdot \hat{N} \cdot f^2)$.

7.2.1 Traditional DM Scheduling

Leung and Whitehead proved that DM is an optimal fixed-priority scheduling algorithm on uniprocessor for constrained-deadline sporadic tasks [LW82]. Necessary and sufficient (exact) schedulability condition for uniprocessor DM scheduling have been derived in [JP86, ABR$^+$93, ABRW91] without considering occurrences of faults. The exact DM schedulability condition proposed in [ABR$^+$93] is derived by assuming that all tasks are released at time 0 (i.e., critical instant for uniprocessor fixed-priority scheduling [LL73]). In [ABR$^+$93], the response-time of each task $\tau_i \in \Gamma$ is given as follows:

$$R_i^{h+1} = C_i + \sum_{j=1}^{i-1} C_j \cdot \left\lceil \frac{R_i^h}{T_j} \right\rceil \tag{7.1}$$

The iteration starts with $R_i^0 = C_i$ and terminates if $R_i^{h+1} = R_i^h$ (schedulable) or $R_i^{h+1} > D_i$ (unschedulable). The exact schedulability test of the entire task set Γ is essentially applying the test in Eq. (7.1) for each task.

The exact analysis as given in Eq (7.1) is not directly applicable for the exact fault-tolerant schedulability analysis of the FTDM scheduling because the worst-case fault pattern considering the assumed fault model, for which the workload within the problem window is maximum, is not known in advance. In this chapter, an exact schedulability condition for FTDM scheduling is derived by computing the exact amount of execution that needs to be completed within the release time and deadline of each task for the assumed fault model.

7.3 Related Work

Many approaches exist in the literature for tolerating faults in real-time tasks. Traditionally, processor failures (permanent faults) are tolerated using Primary and Backup (PB) approach in which the primary and backups of each task are scheduled on two different processors [GMM94, OS94, BMR99, AOSM01, KLLS05b, KLLS05a]. Next chapter deals with algorithm for tolerating permanent processor failures. The discussion of related work for tolerating processor failures is postponed until next chapter.

Ghosh, Melhem and Mossé proposed fault-tolerant uniprocessor scheduling of aperiodic tasks considering transient faults by inserting enough slack in the schedule to allow for the re-execution of tasks when an error is detected [GMM95]. They assumed that the occurrences of two faults are separated by a minimum distance. Pandya and Malek analyzed fault-tolerant RM scheduling on a uniprocessor for tolerating one fault and proved that the minimum achievable utilization bound is 50% [PM98]. The authors also demonstrated the applicability of their scheme for tolerating multiple faults if two faults are separated by a minimum time distance equal to maximum period T_{max} of a task set. In this thesis, the proposed FTDM algorithm can tolerate f task errors within each of all possible time intervals equal to length D_{max} and no restriction is placed in time distance between occurences of two consecutive faults within D_{max}.

Ghosh *et al.* derived a utilization bound for RM uniprocessor scheduling for tolerating single and multiple transient faults using a concept of backup utilization [GMMS98]. To tolerate f transient faults, the utilization of the backup is set to f times the maximum utilization of any task given that a fault model similar to the one in this thesis is used. Such reservation of backup can lead to schedule task sets only having very small total utilization in the fault-free case. Whereas the recovery scheme in [GMMS98] allows backups to execute at a priority higher than that of the faulty task, the recovery scheme in this thesis executes backups at the same priority as the faulty task. Sinha and Suri [SS99] later showed that the proposed protocol in [GMMS98] is in fact faulty.

Liberato, Melhem and Mossé derived both exact and sufficient feasibility conditions for tolerating f transient faults for a set of aperiodic tasks using EDF scheduling [LMM00]. They showed that for a set of n aperiodic tasks in which a maximum of f faults could occur, the exact test can be evaluated in $O(n^2 \cdot f)$ time using a dynamic programming technique. However, the authors of [LMM00] consider backup of a faulty task simply as a re-execution of the primary copy and do not consider the execution of a diverse implementation of a task possibly having a different execution time as backup.

Burns, Davis, and Punnekkat derived an exact fault-tolerant feasibility test for any fixed-priority system using backup that could be simple re-execution or a diverse implementation of the same task [BDP96]. This work is extended in [PBD01] to provide the exact schedulability tests employing check-pointing for fault recovery. In [MdALB03], de A Lima and Burns proposed an optimal fixed-priority assignment to tasks for fault-tolerant scheduling based on re-execution. The fixed priorities of the tasks can be determined in $O(n^2)$ time for a set of n periodic tasks. The schedulability analysis in [BDP96, MdALB03] require the information about the minimum time distance between any two consecutive occurrences of transient faults within the schedule, and only considers simple re-execution or *exactly* one different implementation when an error is detected. In the latter case, the execution time of the backup is the same regardless of the number of errors affecting a particular job. This is in contrast to the proposed method in this thesis where each backup for a particular job may have different execution time.

Based on the *last chance strategy* of Chetto and Chetto [CC89] (in which backups execute at late as possible), software faults are tolerated by considering two versions of each periodic tasks: a primary and a backup [HSW03]. Backups are scheduled as late as possible using a backward RM algorithm (schedule from backward in time). Similar to the work in [MdALB03], the work in [HSW03] considers that there is only one backup for each task and therefore does not have the provision for considering different backups of the same task if more than one fault affect the same task.

Santos *et al.* in [SSO05] derived a schedulability condition for determining the combinations of faults in jobs that can be tolerated using fault-tolerant RM scheduling of periodic tasks. The work in [SSO05] is based a notion, called k-RM schedulable (originally proposed in [SUSO04]). By k-RM schedulable, the authors mean that there are at least k free time slots available between the release time and deadline of each task. In order to guarantee that the system can tolerate multiple transient faults for any combination of faults, all possible fault patterns has to be considered in their derived condition

which gives an intractable time complexity. Moreover, the authors assumed that a fault can occur only in the primary copy of a job.

A fault-burst model is recently defined by Many and Doose in [MD11] as a bounded time interval during which the execution of the tasks are disturbed due to the occurrences of faults for which the distribution of the faults is unknown. Although [MD11] assumes arbitrary number of faults in a fault burst, the proposed recovery strategy in fact considers a finite number of errors to be tolerated within an interval of length D_{max} where only one job of each task is assumed to be faulty. In contrast, the proposed FTDM algorithm considers that multiple jobs of the same task can be disturbed due to burst of faults within an interval of length D_{max}.

Aydin in [Ayd07] proposed aperiodic and periodic task scheduling based on an exact EDF feasibility analysis in which a backup of a task can be different from the primary. Aydin considers a fault model in which a maximum of f transient errors could occur in tasks of the aperiodic task set. The schedulability analysis in [Ayd07] is based on processor demand analysis proposed by Baruah et al. in [BRH90]. For periodic task systems, the proposed exact feasibility test in [Ayd07] is evaluated in $O(\hat{N}^2_{hyper} \cdot f^2_{hyper})$ time, where $\hat{N}_{hyper}$ is the number of jobs released within the first hyper-period (i.e. least common multiple of all the tasks periods) and f_{hyper} is the number of task errors that can occur within the first hyper-period.

In this thesis, the derived exact DM feasibility condition has run-time complexity of $O(n \cdot \hat{N} \cdot f^2)$ where $\hat{N}$ is the maximum number of jobs of the n sporadic tasks released within a time interval of length D_{max}, and f is the maximum number of task errors that can occur within any time interval of length D_{max}. Therefore, the (pseudo-polynomial) time complexity of the proposed exact test is more efficient than the exponential time-complexity of the exact EDF test proposed in [Ayd07].

In summary, most of the work related to developing fault-tolerant scheduling algorithms using time redundancy consider a fault model that is not as general as the fault model considered in this thesis. In many other works, a relatively restricted fault model is considered, assuming, for example, that

- the inter-arrival time of two faults must be separated by a minimum distance [GMM95, PM98, BDP96, MdALB03, PBD01]
- at most one fault may occur in one task [PBD01, HSW03]
- the backup is simply the re-execution of the original task (i.e., does not consider diverse implementation of the task) [GMM95, PM98, PBD01, LMM00, MD11]

7.4 Problem Formulation

The uniprocessor fault-tolerant scheduling algorithm FTDM proposed in this thesis is based on an exact schedulability analysis of the tasks. An occurrences of a maximum of f task errors within each of all possible time intervals of length D_{max} is considered. The f task errors could be distributed over any subset of jobs that are eligible to execute

within the time interval of length D_{max}. Note that a job is eligible to execute between its release time and its deadline. The problem addressed in this chapter is:

> **Is the task set Γ FTDM-schedulable if a maximum of f task errors occur within any time interval of length equal to D_{max}?**

The exact schedulability condition of task set Γ for the fault-tolerant scheduling algorithm FTDM can be derived based on exact feasibility condition of each task $\tau_i \in \Gamma$, for $i = 1, 2, \ldots n$. If a maximum of f task errors can occur within a time interval of length D_{max}, then the maximum number of such errors that can occur within any time interval of length D_i, for $i = 1, 2, 3, \ldots n$, can be at most f. Following this, the last problem statement can be re-written as:

> **Is task τ_i FTDM-schedulable if a maximum of f task errors occur within any time interval of length equal to D_i, for $i = 1, 2, \ldots n$?**

If the exact schedulability condition for each task $\tau_i \in \Gamma$ can be determined, then the exact schedulability condition for the entire task set Γ follows immediately. To ensure that task τ_i is FTDM-schedulable on uniprocessor, the critical instant for which the workload imposed by the higher-priority tasks on task τ_i is maximized needs to be considered in the fault-tolerant schedule. Under the assumed fault model, the critical instant in the uniprocessor fault-tolerant schedule is when all the tasks are released at the same time (as discussed in Section 3.1). In this chapter, without loss of generality, it is assumed that all the tasks are released simultaneously at time zero. In order to derive the exact schedulability condition of task τ_i, it is sufficient to derive the exact schedulability condition for the first job of each task $\tau_i \in \Gamma$. The first job of task τ_i become eligible for execution at time 0 and must finish its execution (including any possible execution of backup due to faults) before time D_i. Consequently, the problem addressed can finally be re-written as:

> **Is the first job of task τ_i FTDM-schedulable if a maximum of f task errors occur within the time interval [0, D_i), for $i = 1, 2, \ldots n$?**

In the rest of this chapter, the exact schedulability condition of task τ_i refers to the exact schedulability condition of the first job of τ_i unless otherwise specified. During the schedulability analysis, the following considerations and assumptions are made:

- The critical instant for each task is at time zero where all the tasks are simultaneously released for the first time.
- Considering the critical instant, the workload within the time interval $[0, D_i)$ is maximized if the jobs of each sporadic task is arrived as quickly as possible (strictly periodic task set).
- Considering the critical instant and strictly periodic releases of the jobs of each task, the job J_i^j of task τ_i is released at time $r_i^j = T_i \cdot (j-1)$ and has its deadline at $d_i^j = r_i^j + D_i$.

- An error is assumed to be detected at the end of execution of the primary or backup. This assumption is necessary for the worst-case schedulability analysis since it corresponds to larger wasted CPU time in comparison to the situation when the error is detected in the middle of execution.
- There is no fault propagation. One fault is assumed to affect at most one job either the primary or the backup. And, any primary or backup is affected by at most one fault since multiple faults affecting the same primary or backup does not cause any increase in recovery workload according to the FTDM scheduling.

The exact schedulability analysis of task τ_i within the interval $[0, D_i)$ is presented in Section 7.5. In order to find the worst-case workload required to be completed within an interval $[0, D_i)$ on behalf of the higher priority sporadic tasks, it is not difficult to see that the work within the interval is maximized under the assumption that the jobs of the tasks arrive as quickly as possible (as is assumed above). In order to find the exact schedulability condition, the maximum total work completed within $[0, D_i)$ by the jobs of the tasks $\{\tau_1, \tau_2 \ldots \tau_i\}$ is calculated based on two *load factors*.

In subsection 7.5.1, the first load factor that is equal to the maximum work that needs to be completed by a job of task τ_i in $[0, D_i)$ is calculated. Then in subsection 7.5.2, the second load factor that is equal to the maximum work that need to be completed within $[0, D_i)$ by the higher priority jobs of the tasks $\{\tau_1, \tau_2 \ldots \tau_{i-1}\}$ is calculated. This second load factor is calculated as follows. First, the different subsets of higher priority jobs such that all the jobs in each such subset are released at the same time at some time instant within $[0, D_i)$ are determined. Then, based on each of these different subsets, the execution requirement of all the higher-priority jobs is abstracted by means of two *composition* techniques, called *vertical composition* and *horizontal composition*, to find the maximum work completed by the higher priority jobs within $[0, D_i)$ in subsection 7.5.2.

7.5 Load Factors and Composability

In this section, the fundamental theoretical building blocks for the schedulability analysis of task τ_i within the time interval $[0, D_i)$ in terms of load factors and compositions are derived. To determine whether the first job of task τ_i is schedulable, the amount of execution completed by higher-priority jobs within $[0, D_i)$ needs to be calculated. Note that the maximum amount of execution completed by the higher-priority jobs depends on different fault patterns affecting these higher-priority jobs. By subtracting the maximum amount of execution completed by the higher- priority jobs within $[0, D_i)$ from D_i, the maximum available time for execution of task τ_i within $[0, D_i)$ can be derived. To determine whether the available execution time for task τ_i is enough for its complete execution within $[0, D_i)$, it is needed to know the maximum amount of execution required to be completed by the first job of task τ_i. This amount of execution depends on the number of task errors exclusively affecting task τ_i within $[0, D_i)$.

When analyzing the schedulability of τ_i, the *worst-case workload* within $[0, D_i)$ is the maximum execution completed by the jobs of the tasks in set $\{\tau_1, \tau_2 \ldots \tau_i\}$ that

are released within $[0, D_i)$. Remember that at most f task errors could occur within $[0, D_i)$. To find this worst-case workload required to be completed within $[0, D_i)$ by the jobs of the tasks in set $\{\tau_1, \tau_2 \ldots \tau_i\}$, one has to consider (i) the occurrences of k task errors affecting the jobs of the higher-priority tasks (including their backups), and (ii) the occurrences of $(f - k)$ task errors exclusively affecting the first job of task τ_i and its backups, for $k = 0, 1, 2, \ldots f$. In summary, to find the worst-case workload within $[0, D_i)$, the following two workload factors are determined:

1. `Load-Factor-i`: Execution time required by task τ_i when $(f - k)$ errors exclusively affect the first job of task τ_i, for $k = 0, 1, 2 \ldots f$.

2. `Load-Factor-HPi`: Execution time required by the higher-priority jobs within $[0, D_i)$ when k errors affect these higher-priority jobs in this interval, for $k = 0, 1, 2 \ldots f$.

The worst-case workload within $[0, D_i)$ can now be defined as the sum of these two load factors such that this sum is maximized for some k, $0 \leq k \leq f$. To meet the deadline of task τ_i, the complete execution of task τ_i (including the execution of its backups) must take place within the interval $[0, D_i)$. However, parts of the execution of jobs released within $[0, D_i)$ and having higher priority than the priority of task τ_i may take place outside the interval $[0, D_i)$. If the execution of any higher-priority job takes place outside the interval $[0, D_i)$, the execution time beyond time instant D_i must not be accounted in the calculation of `Load-Factor-HPi`. This is to avoid overestimating the amount of worst-case workload within the interval $[0, D_i)$ and to derive an exact schedulability test for `FTDM` scheduling.

If the sum of `Load-Factor-i` and `Load-Factor-HPi`, i.e., the maximum workload in $[0, D_i)$, is not greater than D_i, then task τ_i has enough time to finish its complete execution within $[0, D_i)$. Thus, based on the values of the two workload factors, the exact schedulability condition for task τ_i is derived in this thesis. The calculation of the two workload factors (that is, value of `Load-Factor-i` and `Load-Factor-HPi`) are presented in subsection 7.5.1 and subsection 7.5.2, respectively.

7.5.1 Calculation of `Load-Factor-i`

The value of `Load-Factor-i` is the execution time required by task τ_i when $(f - k)$ task errors exclusively affect task τ_i, for $k = 0, 1, 2 \ldots f$. If an error is detected after executing of the primary of the first job task τ_i, then the first backup of task τ_i is ready for execution. If an error is detected at the end of execution of a backup of task τ_i, then the next backup of task τ_i is ready for execution. Remember that the WCET of the b^{th} backup of task τ_i is denoted by E_i^b, for $b = 1, 2 \ldots f$. The total execution time required due to the $(f-k)$ errors affecting the primary and backups of a particular job of task τ_i is denoted by $C_i^{(f-k)}$. The value of `Load-Factor-i` is equal to $C_i^{(f-k)}$ and has to be calculated for all $k = 0, 1, 2, \ldots f$. The value of $C_i^{(f-k)}$ can be recursively calculated

using Eq. (7.2) as follows:

$$C_i^{(f\,-\,k)} = \begin{cases} C_i & \text{if } (f-k) = 0 \\ E_i^{(f-k)} + C_i^{(f\,-\,k\,-\,1)} & \text{if } (f-k) > 0 \end{cases} \tag{7.2}$$

The value of $C_i^{(f\,-\,k)}$ is set equal to C_i when $(f-k)$ is equal to 0. When $(f-k)$ is equal to 0, only the execution time of the primary copy of task τ_i is considered in Eq. (7.2). In the recursive part of Eq. (7.2), the execution time of the $(f-k)^{th}$ backup of task τ_i and the execution time due to a total of $(f-k-1)$ task errors affecting task τ_i are added to find the value of $C_i^{(f\,-\,k)}$. Using Eq. (7.2), starting from $k = f, (f-1), \ldots 0$, the value $C_i^{(f\,-\,k)}$ can be calculated for all $(f-k) = 0, 1, 2 \ldots f$ using a total of $O(f)$ addition operations. The task τ_i must complete $C_i^{(f\,-\,k)}$ units of execution within the interval $[0, D_i)$ to tolerate $(f-k)$ task errors that exclusively affect the first job of task τ_i. The calculation of `Load-Factor-i` is now demonstrated using an example.

Example 7.1. *Consider a task set {τ_1, τ_2, τ_3} given in Table 7.1 for f=2. The first column in Table 7.1 represents the name of each task. The second and third columns represent the relative deadline and period of each task, respectively. The WCET of the primary copy of each task is given in the fourth column. The fifth and sixth columns represent the WCET of the first and second (since $f = 2$, at most two errors can occur in the same job of any task for the assumed fault model) backups of each task, respectively. Note that the WCET of a backup of a task may be equal to, greater or smaller than the WCET of the primary of the corresponding task. Using Eq.* (7.2)*, the amount of*

τ_i	D_i	T_i	C_i	E_i^1	E_i^2
τ_1	10	10	3	2	3
τ_2	15	15	3	4	2
τ_3	40	40	9	8	6

Table 7.1: *Example task set with f=2 backups for each task*

execution time required for each task τ_i due to $(f-k)$ task errors exclusively affecting task τ_i is calculated in Eq. (7.3) *for $k = 0, 1, 2$ and $f = 2$ as follows:*

For task τ_1,	*For task τ_2,*	*For task τ_3,*
$C_1^0 = C_1 = 3$	$C_2^0 = C_2 = 3$	$C_3^0 = C_3 = 9$
$C_1^1 = E_1^1 + C_1^0 = 5$	$C_2^1 = E_2^1 + C_2^0 = 7$	$C_3^1 = E_3^1 + C_3^0 = 17$
$C_1^2 = E_1^2 + C_1^1 = 8$	$C_2^2 = E_2^2 + C_2^1 = 9$	$C_3^2 = E_3^2 + C_3^1 = 23$

(7.3)

The task set in Table 7.1 is used in the rest of this chapter as the running example. The calculation of the value of `Load-Factor-HPi` is presented in next subsection.

7.5.2 Calculation of `Load-Factor-HPi`

The value of `Load-Factor-HPi` is the maximum execution time completed within $[0, D_i)$ by the jobs having higher priority than the priority of task τ_i, when k errors affect these higher-priority jobs within $[0, D_i)$. If the execution of some of these higher-priority jobs takes place outside $[0, D_i)$, then only the execution that takes place within $[0, D_i)$ must be considered in the calculation of `Load-Factor-HPi`. This is a very crucial issue in determining the value of `Load-Factor-HPi`, as can be seen in the following example.

Example 7.2. *Consider the first job of task τ_2 in Table 7.1 that is to be scheduled within the interval $(0, 15]$ since $D_2 = 15$. Assume that jobs of the only higher priority task τ_1 are released as soon as possible: J_1^1 and J_1^2 are the jobs that are released within the interval $[0, 15)$ and have higher priority than the priority of task τ_2. The primary of each of the jobs J_1^1 and J_1^2 executes within the interval $[0, 3)$ and $[10, 13)$, respectively.*

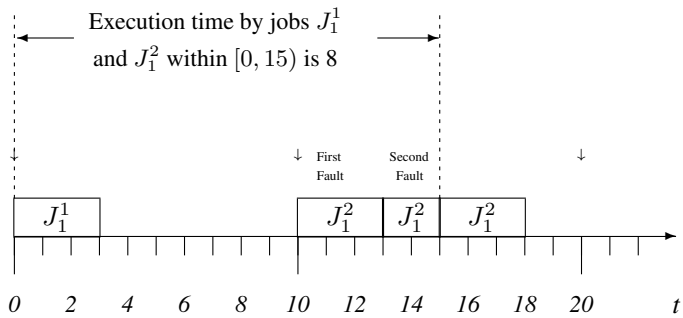

Figure 7.1: *Schedule of jobs J_1^1 and J_1^2. The downward vertical arrows denotes the arrival time of the jobs of τ_1. The two errors occur in the primary and the first backup of job J_1^2. The maximum amount of total execution by the jobs J_1^1 and J_1^2 due to the two errors is equal to 11. However, the amount of maximum total execution by the jobs J_1^1 and J_1^2 within the interval $[0, 15)$ is 8, not 11.*

Now, consider a 2-fault pattern in which the first and the second errors affect the primary and the first backup of job J_1^2, respectively. The detection of the second error in the first backup of job J_1^2 triggers the execution of the second backup of job J_1^2. The first and second backups of job J_1^2 executes within the interval $[13, 15)$ and $[15, 18)$, respectively. The schedule of the jobs J_1^1 and J_1^2 including the execution of the backups for the considered 2-fault pattern is shown in Figure 7.1. The total execution time required by the higher-priority jobs J_1^1 and J_1^2 is $(3 + 3 + 2 + 3) = 11$ time unit (including time for recovery). Notice that, the second backup of job J_1^2 executes outside the interval $[0, D_2)$. The maximum execution time by the jobs J_1^1 and J_1^2 within the interval $[0, D_2)$ is equal to $(3 + 3 + 2) = 8$, not 11 for the considered 2-fault pattern. □

When calculating the worst-case workload in $[0, D_i)$ to derive the exact `FTDM` schedulability test of task τ_i, the value of `Load-Factor-HPi` must not be overestimated. To

calculate the value of `Load-Factor-HPi`, the jobs that are released within interval $[0, D_i)$ and have higher priority than the priority of task τ_i need to be determined. The set of jobs having higher-priority than the priority of task τ_i is denoted by a set $\mathtt{HPJ}_i$ such that each job in set $\mathtt{HPJ}_i$ is released within the interval $[0, D_i)$. That is, the set $\mathtt{HPJ}_i$ is defined in Eq. (7.4) as follows:

$$\mathtt{HPJ}_i = \{J_p^q \mid p < i \text{ and } r_p^q < D_i\} \tag{7.4}$$

where $r_p^q = T_p \cdot (q-1)$ and $q = 1, 2, \ldots$. According to Eq. (7.4), if job $J_p^q \in \mathtt{HPJ}_i$, then task τ_p has shorter deadline (that is, higher priority[1]) than task τ_i and the release time of job J_p^q (that is, value of r_p^q defined in Eq. (3.1)) is less than D_i. Each of the higher-priority jobs in set $\mathtt{HPJ}_i$ is eligible for execution at or after its release time within $[0, D_i)$. In the case of our running example, the sets $\mathtt{HPJ}_i$ for $i = 1, 2, 3$ are determined for the three tasks in Table 7.1.

Example 7.3. *Using Eq.* (7.4) *for the task set in Table 7.1 we have,*

$$\begin{aligned}
&[0, D_1) = [0, 9) \quad \textit{and} \quad \mathtt{HPJ}_1 = \emptyset \\
&[0, D_2) = [0, 15) \quad \textit{and} \quad \mathtt{HPJ}_2 = \{J_1^1\ ,\ J_1^2\ \} \\
&[0, D_3) = [0, 20) \quad \textit{and} \quad \mathtt{HPJ}_3 = \{J_1^1\ ,\ J_1^2\ ,\ J_1^3\ ,\ J_1^4\ ,\ J_2^1\ ,\ J_2^2\ ,\ J_2^3\ \}
\end{aligned} \tag{7.5}$$ □

Remember that $\hat{N}$ is the maximum number of jobs that are released within the time interval $[0, D_{max})$. Therefore, the number of jobs having higher priority than the priority of task τ_i that are released within $[0, D_i)$ is at most $\hat{N}$. If the release time of a higher-priority job J_p^q is earlier than D_i, then J_p^q is included in $\mathtt{HPJ}_i$. Therefore, the time complexity to find the set $\mathtt{HPJ}_i$ is $O(\hat{N})$.

When considering the `FTDM` schedulability of the first job of task τ_i, the value of `Load-Factor-HPi` for a k-fault pattern such that the k errors affect the jobs in set $\mathtt{HPJ}_i$ needs to be calculated for $k = 0, 1, \ldots f$. The value of `Load-Factor-HPi` is a measure of how much computation is completed within the interval $[0, D_i)$ by the higher-priority jobs in set $\mathtt{HPJ}_i$ due to the k-fault pattern. The amount of computation *completed* by the jobs in set $\mathtt{HPJ}_i$ within $[0, D_i)$ depends on how much workload is *requested* by the jobs in $\mathtt{HPJ}_i$ due to the k-fault pattern. Aydin in [Ayd07] used a dynamic programming technique to compute the maximum workload requested by a set of aperiodic tasks due to a k-fault pattern. Using an approach similar to that in [Ayd07], the maximum workload requested by a set of higher-priority jobs that are all released at a particular time instant t within the time interval $[0, D_i)$ is computed.

The maximum workload requested by a set of jobs in set A, all released at a particular time instant t, is denoted by function $L_k(A)$ for a k-fault pattern[2]. Note that the value of $L_k(A)$ is the maximum workload requested by the jobs in set A, not the actual

[1] Ties between the deadlines of two tasks can be broken arbitrarily.

[2] The jobs in set A are released at time t. The time instant t is not included in function $L_k(A)$ and can be understood from the context. Although the value of $L_k(A)$ can be calculated independent of t, the context t is important for the schedulability analysis as will be evident shortly.

amount of execution by the jobs in set A within $[0, D_i)$ because some of the workload may need to be executed after $[0, D_i)$. The function $L_k(A)$ is defined recursively (similar to [Ayd07], but the difference being that all the jobs in set A have the same release time) in Eq. (7.6) and Eq. (7.7). The basis of the recursion is defined in Eq. (7.6) considering exactly one job J_x^y exists in set A, for $k = 0, 1, 2, \ldots f$, as follows

$$L_k(\{J_x^y\}) = C_x^k \tag{7.6}$$

The value of $L_k(\{J_x^y\})$ represents the amount of execution time requested by job J_x^y when k errors exclusively affect the primary and backups of job J_x^y. Remember that the value of C_x^k is defined in Eq. (7.2) as the maximum amount of execution time required by the task τ_x when k errors exclusively affect a particular job of this task. The value of C_x^k in the right hand side of Eq. (7.6) can be calculated using Eq. (7.2) in $O(f)$ time, for all $k = 0, 1, 2 \ldots f$.

By assuming that the value of $L_k(A)$ is known, the value of $L_k(\mathrm{A} \cup \{J_x^y\})$ is computed recursively, for $k = 0, 1, 2 \ldots f$, as follows:

$$L_k(\mathrm{A} \cup \{J_x^y\}) = \max_{q=0}^{k} \left\{ L_q(A) + L_{k-q}(\{J_x^y\}) \right\} \tag{7.7}$$

In Eq. (7.7), the value of $L_k(\mathrm{A} \cup \{J_x^y\})$ is maximum for one of the $(k+1)$ possible values of q, where $0 \leq q \leq k$, for the right hand side of Eq. (7.7). The value of q is selected such that, if q errors occur in the jobs in set A and $(k-q)$ errors occur exclusively in job J_x^y, then $L_k(\mathrm{A} \cup \{J_x^y\})$ is at its maximum for some q, $0 \leq q \leq k$. The working of Eq. (7.7) is now demonstrated using an example.

Example 7.4. *Consider the lowest-priority task τ_3 given in Table 7.1. The jobs, having higher priority than the priority of task τ_3, that are released at time $t = 0$ are in the set A={J_1^1, J_2^1}. To determine the maximum workload requested by the higher-priority jobs in set A={J_1^1, J_2^1} due to a k-fault pattern, one needs to calculate the value of $L_k(A)$. To calculate $L_k(A)$, the base in Eq.* (7.6) *for each of the jobs in set A need to be computed considering the occurrences of k errors exclusively affecting that job. Since f is equal to 2, the possible values of k are 0, 1 and 2.*

According to Eq. (7.3)*, the maximum execution time required for job J_1^1 is C_1^0 =3, C_1^1 =5 and C_1^2 =8 for $k = 0$, $k = 1$ and $k = 2$ errors exclusively affecting job J_1^1, respectively. The maximum execution time required for job J_2^1 is C_2^0 =3, C_2^1 =7 and C_2^2 =9 for $k = 0$, $k = 1$ and $k = 2$ errors exclusively affecting job J_2^1, respectively (according to Eq.* (7.3)*). Using the base of the recursion in Eq.* (7.6)*, we have*

$$L_0(\{J_1^1\}) = C_1^0 = 3 \quad L_1(\{J_1^1\}) = C_1^1 = 5 \quad L_2(\{J_1^1\}) = C_1^2 = 8$$

$$L_0(\{J_2^1\}) = C_2^0 = 3 \quad L_1(\{J_2^1\}) = C_2^1 = 7 \quad L_2(\{J_2^1\}) = C_2^2 = 9$$

Using Eq. (7.7)*, the value of $L_k(A)$ for $k = 0, 1, 2$ and A={J_1^1,J_2^1} can be calculated*

as follows:

$$\begin{aligned}
L_0(\{J_1^1\ ,\ J_2^1\ \}) &= \max_{q=0}^{0} \left\{L_q(\{J_1^1\ \}) + L_{0-q}(\{J_2^1\})\right\} \\
&= L_0(\{J_1^1\ \}) + L_0(\{J_2^1\}) \\
&= 3+3=6 \\
L_1(\{J_1^1\ ,\ J_2^1\ \}) &= \max_{q=0}^{1} \left\{L_q(\{J_1^1\ \}) + L_{1-q}(\{J_2^1\})\right\} \\
&= max\left\{\ L_0(\{J_1^1\ \}) + L_1(\{J_2^1\})\ ,\right. \\
&\qquad \left. L_1(\{J_1^1\ \}) + L_0(\{J_2^1\})\right\} \\
&= max\left\{3+7, 5+3\right\} = 10 \\
L_2(\{J_1^1\ ,\ J_2^1\ \}) &= \max_{q=0}^{2} \left\{L_q(\{J_1^1\ \}) + L_{2-q}(\{J_2^1\})\right\} \\
&= max\left\{\ L_0(\{J_1^1\ \}) + \ L_2(\{J_2^1\})\ ,\right. \\
&\qquad L_1(\{J_1^1\ \}) + L_1(\{J_2^1\})\ , \\
&\qquad \left. L_2(\{J_1^1\ \}) + L_0(\{J_2^1\})\right\} \\
&= max\left\{3+9, 5+7, 8+3\right\} = 12
\end{aligned}$$

The maximum amount of workload requested by the jobs in set A=$\{J_1^1, J_2^1\}$ is $L_0(A)$=6, $L_1(A)$=10, and $L_2(A)$=12 for $k=0,1$ and 2-fault-patterns, respectively. □

Time complexity to calculate $L_k(\mathbf{A} \cup \{J_x^y\})$: There are $(|A|+1)$ jobs in set $(A \cup \{J_x^y\})$. For each one of the $(|A|+1)$ jobs, evaluating the base case using Eq. (7.6) can be done using Eq. (7.2) in $O(f)$ steps for all $k = 0, 1, 2, \ldots f$. Therefore, evaluating the base for all the jobs in set $(A \cup \{J_x^y\})$ requires $[(|A|+1) \cdot O(f)] = O(|A| \cdot f)$ operations.

For the recursive step, if the value of $L_k(A)$ is known, then there are $(k+1)$ possibilities for the selection of q in Eq. (7.7) to compute $L_k(\mathrm{A} \cup \{J_x^y\})$ for a given k, $0 \leq k \leq f$. Therefore, computing $L_k(\mathrm{A} \cup \{J_x^y\})$ requires $O(k)$ operations ($k+1$ additions and k comparisons) for a particular k and given that $L_k(A)$ is known. Given that the values of $L_k(A)$ are known for all $k = 0, 1, 2, \ldots f$, then computing $L_k(\mathrm{A} \cup \{J_x^y\})$ for all $k = 0, 1, \ldots f$ requires total $O(0+1+2 \ldots f) = O(f^2)$ operations.

Starting with one job in set A, a new job J_x^y is considered when computing the value of $L_k(\mathrm{A} \cup \{J_x^y\})$. By including one job J_x^y in the set A at each step, the set $(A \cup \{J_x^y\})$ is finally formed. Therefore, for all the jobs in the set $(A \cup \{J_x^y\})$, the total time complexity to recursively compute the value of $L_k(\mathrm{A} \cup \{J_x^y\})$ is equal to $[(|A|+1) \cdot O(f^2)] = O(|A| \cdot f^2)$. Consequently, the total time complexity for the base and recursive steps to compute $L_k(\mathrm{A} \cup \{J_x^y\})$ is $O(|A| \cdot f + |A| \cdot f^2) = O(|A| \cdot f^2)$. □

As mentioned before, the value of `Load-Factor-HPi` is the maximum execution completed within the interval $[0, D_i)$ by the jobs having higher priorities than the priority of task τ_i for a k-fault pattern. The maximum execution completed by the set of higher-priority jobs within $[0, D_i)$ may not be same as the maximum workload requested by

this set of higher-priority jobs for a k-fault pattern.

Remember that the value of $L_k(A)$ is calculated considering that all the jobs in set A are released at the same time, say at time t. Consider that the set A contains the jobs having higher priority than the priority of task τ_i and all the jobs in set A are released at time t. If the value of $L_k(A)$ is greater than $(D_i - t)$, then the maximum amount of work completed by the higher-priority jobs in set A within the interval $[0, D_i)$ is at most $(D_i - t)$ using the work-conserving algorithm FTDM. If $L_k(A)$ is less than or equal to $(D_i - t)$, then the maximum amount of work that can be completed by the jobs in set A within the interval $[0, D_i)$ is at most $L_k(A)$. This crucial observation is later used to compose the workload of the higher priority jobs within the interval $[0, D_i)$.

In order to find the amount of execution completed by the jobs of the higher-priority tasks within the time interval $[0, D_i)$, the higher-priority jobs released at different time instants within the time interval $[0, D_i)$ are *composed*. A composed task is not an actual task in the system rather a way to represent the execution of a collection of higher-priority jobs in a compact (composed) way. The execution time of a composed task (formally defined later) represents the maximum amount of execution within the interval $[0, D_i)$ if the jobs represented by the composed tasks have exclusive access to the processor within the interval $[0, D_i)$. In other words, the execution time of a composed task is the amount of maximum execution within the interval $[0, D_i)$ if only the jobs represented by the composed task are allowed to execute within the interval $[0, D_i)$.

The composition of the higher-priority tasks are done in two steps: first by *vertical composition* and then by *horizontal composition*. Each vertically-composed task abstracts the higher-priority jobs that are all released at a particular time instant within $[0, D_i)$. Each horizontally-composed task abstracts the higher-priority jobs that are abstracted by more than one vertically-composed task. Horizontal composition is presented next following vertical composition.

Vertical Composition

Consider a set of all jobs that are released at time instant t, $t < D_i$ and have higher priority than the priority of task τ_i. To compactly represent these higher-priority jobs, a vertically-composed task, denoted by $V_{\{t\}}$, is defined such that the composed task $V_{\{t\}}$ abstracts the set of higher-priority jobs that are all released at time t where $0 \leq t < D_i$. The execution time of the composed task $V_{\{t\}}$ (formally calculated later) denotes the maximum amount of execution that can be completed within $[0, D_i)$ by the higher-priority jobs that are released at time t such that only the jobs represented by $V_{\{t\}}$ are allowed to execute within $[0, D_i)$. One vertically-composed task is formed for each time instant within $[0, D_i)$ at which new higher-priority jobs are released.

Example 7.5. *Consider the schedulability of task τ_3 in Table 7.1. The first job of task τ_3 is released at time 0 and has its deadline by time $D_3 = 40$. The tasks τ_1 and τ_2 are the higher-priority tasks of τ_3. The releases of the higher-priority jobs at different time instants within the interval $[0, 40)$ is shown in Figure 7.2 using downward arrows by assuming strictly periodic arrival of the jobs.*

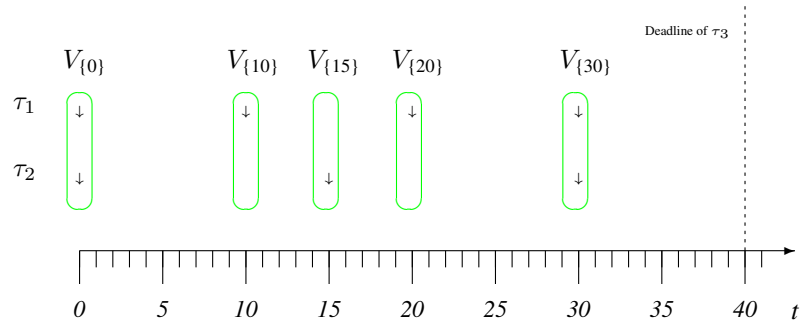

Figure 7.2: *Five vertically-composed tasks are shown using vertically long ovals at time instants 0, 10, 15, 20, and 30. Each vertically-composed task at time t abstracts all the newly released higher-priority jobs of task τ_3 that are released at time t within the time interval* $[0, 40)$.

New jobs of the higher-priority tasks are released at time instants 0, 10, 15, 20 and 30. At each of these five time instants, a vertically-composed task is formed (that abstracts the released jobs shown in each oval in Figure 7.2). The five composed tasks are denoted by $V_{\{0\}}$,$V_{\{10\}}$,$V_{\{15\}}$,$V_{\{20\}}$ and $V_{\{30\}}$ in Figure 7.2. □

To form the vertically-composed tasks, the different time points in $[0, D_i)$ where new jobs of the higher-priority tasks are released need to be determined. The set of time points, denoted by S_i, where jobs having higher priority than the priority of task τ_i are released within the interval $[0, D_i)$ is given by Eq. (7.8) as follows:

$$S_i = \{k \cdot T_j \mid j = 1 \ldots (i-1), k = 0 \ldots \left\lfloor \frac{D_i}{T_j} \right\rfloor\} - \{D_i\} \tag{7.8}$$

Each of the time points in set S_i are less than D_i and are nonnegative integer multiples of the periods of the higher-priority task τ_j for $j = 1, 2, \ldots (i-1)$ assuming the critical instant (i.e., all the tasks first arrives at time 0). Since the higher-priority jobs released at or beyond time instant D_i will not execute prior to time instant D_i, it is necessary that all the time points in set S_i are less than D_i (that is, before the deadline of the first job of task τ_i). At each of the time points in set S_i, new higher-priority jobs are released by assuming that jobs of the higher priority tasks are released as quickly as possible.

Example 7.6. *Consider the task set given in Table 7.1. Using Eq.* (7.8)*, we have*

$$\begin{aligned} S_1 &= \{\} \\ S_2 &= \{0, 10\} - \{15\} = \{0, 10\} \\ S_3 &= \{0, 10, 15, 20, 30, 40\} - \{40\} = \{0, 10, 15, 20, 30\} \end{aligned} \tag{7.9}$$

□

The jobs having higher priorities than that of task τ_i are released at each of the time points in set S_i. Remember that there are at most $\hat{N}$ jobs released within any interval of length D_{max}. The time points in S_i are integer multiples of the periods of the higher-priority tasks. Therefore, the run-time complexity to compute S_i is $O(\hat{N})$.

During the schedulability analysis of task τ_i, we have to consider each time point in set S_i where some new higher-priority jobs of task τ_i are released. For each $s \in S_i$, a vertically-composed task $V_{\{s\}}$ is formed. In the case of the example in Table 7.1, when analyzing the schedulability of task τ_3, one vertically-composed task for each $s \in S_3 = \{0,10,15,20,30\}$ is formed (see the five vertically-composed tasks in Figure 7.2).

The vertically-composed task $V_{\{s\}}$ for $s \in S_i$ abstracts the set of higher-priority jobs from set $\mathtt{HPJ}_i$ that are all released at time s. To find the execution time of a vertically-composed task at time $s \in S_i$, the higher-priority jobs in set $\mathtt{HPJ}_i$ that are released at time instant s need to be determined. The set $\mathtt{Rel}_{i,s}$ denotes the higher-priority jobs of task τ_i that are released at time s. The set $\mathtt{Rel}_{i,s}$ is given in Eq. (7.10) as follows:

$$\mathtt{Rel}_{i,s} = \{J_p^q \mid J_p^q \in \mathtt{HPJ}_i \text{ and } r_p^q = s\} \tag{7.10}$$

The set $\mathtt{Rel}_{i,s}$ contains the jobs that are released at time s and are of higher priority than task τ_i. If job J_p^q is in set $\mathtt{Rel}_{i,s}$, then job J_p^q is in set $\mathtt{HPJ}_i$ and the release time of job J_p^q is equal to time instant s, that is, s is equal to r_p^q. The condition in Eq. (7.10) is to be evaluated for each job in set $\mathtt{HPJ}_i$. Since there are at most $\hat{N}$ jobs released within any time interval of length D_{max}, the number of jobs in set $\mathtt{HPJ}_i$ is $O(\hat{N})$. The job $J_p^q \in \mathtt{HPJ}_i$ is stored in set $\mathtt{Rel}_{i,s}$ if the release time r_p^q is equal to s. By selecting one by one job J_p^q from set $\mathtt{HPJ}_i$, the job J_p^q can be stored in the appropriate set $\mathtt{Rel}_{i,s}$ such that the release time r_p^q of job J_p^q is equal to s. Therefore, the time complexity to find $\mathtt{Rel}_{i,s}$ for all $s \in S_i$ is equal to $O(\hat{N})$.

Example 7.7. *Consider the example task set in Table 7.1. Since there are no higher-priority jobs of task τ_1, the set $\mathtt{HPJ}_1 = \emptyset$. For tasks τ_2 and τ_3 we have $S_2 = \{0, 10\}$ and $S_3 = \{0, 10, 15, 20, 30\}$, respectively, according to Eq.* (7.9)*. The set, $\mathtt{Rel}_{i,s}$, of higher-priority jobs released at different time instant $s \in S_i$ for $i = 2$ and $i = 3$ are given in Eq.* (7.11) *as follows:*

$$\begin{array}{llll} \mathtt{Rel}_{2,0} = \{J_1^1\} & \mathtt{Rel}_{2,10} = \{J_1^2\} \\ \mathtt{Rel}_{3,0} = \{J_1^1, J_2^1\} & \mathtt{Rel}_{3,10} = \{J_1^2\} \\ \mathtt{Rel}_{3,15} = \{J_2^2\} & \mathtt{Rel}_{3,20} = \{J_1^3\} \\ \mathtt{Rel}_{3,30} = \{J_2^3, J_1^4\} \end{array} \tag{7.11}$$

□

The jobs in set $\mathtt{Rel}_{i,s}$ are of higher priority than that of the task τ_i and all these higher-priority jobs are released at time s. For each $s \in S_i$, the vertically-composed task $V_{\{s\}}$ abstracts the jobs in set $\mathtt{Rel}_{i,s}$. What follows next is the technique to calculate the execution time of a vertically-composed task $V_{\{s\}}$.

The execution time of the vertically-composed task $V_{\{s\}}$ is denoted by the function `w(k,{s})` for a k-fault pattern only affecting the jobs in set $\mathtt{Rel}_{i,s}$. If no jobs other than the jobs in set $\mathtt{Rel}_{i,s}$ are allowed to execute within the interval $[0, D_i)$, then the value of `w(k,{s})` represents the maximum amount of execution that can be completed by the jobs in set $\mathtt{Rel}_{i,s}$ within the interval $[0, D_i)$ for a k-fault pattern.

The value of $L_k(\texttt{Rel}_{i,s})$ is the maximum amount of workload requested by the jobs abstracted by the vertically-composed task $V_{\{s\}}$. The set of jobs released at time s can complete, using work conserving algorithm FTDM, at most $(D_i - s)$ amount of work within $[0, D_i)$ if $L_k(\texttt{Rel}_{i,s})$ is greater than $(D_i - s)$. Otherwise, the maximum amount of work completed by the set of jobs released at time s is $L_k(\texttt{Rel}_{i,s})$. To this end, the execution time of $V_{\{s\}}$ for $k = 0, 1, 2, \ldots f$ is defined in Eq. (7.12) as follows:

$$\texttt{w(k,\{s\})} = min\ \{L_k(\texttt{Rel}_{i,s})\ ,\ (D_i\ -s)\ \} \tag{7.12}$$

The value w(k,{s}) represents the maximum amount of execution completed by the jobs released at time s within the interval $[0, D_i)$ if no jobs other than the jobs in set $\texttt{Rel}_{i,s}$ are allowed to execute within the interval $[0, D_i)$. The calculation of w(k,{s}) is shown next for the running example.

Example 7.8. *Consider the task set in Table 7.1. When considering the schedulability of task τ_1, there is no higher-priority jobs of task τ_1. Therefore, no vertically-composed task is formed since set S_1 is empty.*

For $s = 0$ and $k = 0$	For $s = 10$ and $k = 0$
w(0,{0})	w(0,{10})
$= min\{L_0(\texttt{Rel}_{2,0}), D_i\ - 0\}$	$= min\{L_0(\texttt{Rel}_{2,10}), D_i\ - 10\}$
$= min\{L_0(\texttt{Rel}_{2,0}), 15 - 0\}$	$= min\{L_0(\texttt{Rel}_{2,10}), 15 - 10\}$
$= min\{L_0(\{\ J_1^1\ \}), 15\} = min\{3, 15\} = 3$	$= min\{L_0(\{\ J_1^2\ \}), 5\} = min\{3, 5\} = 3$
For $s = 0$ and $k = 1$	For $s = 10$ and $k = 1$
w(1,{0})	w(1,{10})
$= min\{L_1(\texttt{Rel}_{2,0}), D_i\ - 0\}$	$= min\{L_1(\texttt{Rel}_{2,10}), D_i\ - 10\}$
$= min\{L_1(\texttt{Rel}_{2,0}), 15 - 0\}$	$= min\{L_1(\texttt{Rel}_{2,10}), 15 - 10\}$
$= min\{L_1(\{\ J_1^1\ \}), 15\} = min\{5, 15\} = 5$	$= min\{L_1(\{\ J_1^2\ \}), 5\} = min\{5, 5\} = 5$
For $s = 0$ and $k = 2$	For $s = 10$ and $k = 2$
w(2,{0})	w(2,{10})
$= min\{L_2(\texttt{Rel}_{2,0}), D_i\ - 0\}$	$= min\{L_2(\texttt{Rel}_{2,10}), D_i\ - 10\}$
$= min\{L_2(\texttt{Rel}_{2,0}), 15 - 0\}$	$= min\{L_2(\texttt{Rel}_{2,10}), 15 - 10\}$
$= min\{L_2(\{\ J_1^1\ \}), 15\} = min\{8, 15\} = 8$	$= min\{L_2(\{\ J_1^2\ \}), 5\} = min\{8, 5\} = 5$

Table 7.2: *Calculation of w(k,{s}) for vertical composition at each $s \in S_2$ for $k = 0, 1, 2$. The left column show the execution time w(k,{0}) of the vertically-composed task $V_{\{0\}}$ for $k = 0, 1, 2$ faults and the right column show the execution time w(k,{10}) of the vertically-composed task $V_{\{10\}}$ for $k = 0, 1, 2$ faults.*

When considering the schedulability of task τ_2, there are higher-priority jobs that are released within $[0, D_2)$. To find the vertical compositions of the higher-priority jobs,

the following information is used:

$$S_2 = \{0, 10\} \quad \textit{from Eq. (7.9)}$$
$$D_2 = 15 \quad \textit{from Table 7.1}$$
$$\texttt{Rel}_{2,0} = \{\, J_1^1 \,\} \quad \textit{for } s = 0 \textit{ from Eq. (7.11)}$$
$$\texttt{Rel}_{2,10} = \{\, J_1^2 \,\} \quad \textit{for } s = 10 \textit{ from Eq. (7.11)}$$

Two vertically-composed tasks are formed since there are two time points in set $S_2 = \{0, 10\}$. The two vertically-composed tasks are $V_{\{0\}}$ and $V_{\{10\}}$. For each vertically-composed task, the amount of execution time in $[0, D_2)$ can be determined for $k = 0, 1, 2$ (since $f = 2$) using Eq. (7.12). The value of `w(k,{s})` *for the composed task $V_{\{s\}}$ using Eq. (7.12) is calculated in Table 7.2 for $k = 0, 1, 2$ and $s = 0, 10$.*

When considering the schedulability of task τ_3, there are higher-priority jobs that are eligible for execution within $[0, D_3)$. To find the vertical compositions of the higher-priority jobs, the following information is used:

$$S_3 = \{0, 10, 15, 20, 30\} \quad \textit{from Eq. (7.9)}$$
$$D_3 = 40 \quad \textit{from Table 7.1}$$
$$\texttt{Rel}_{3,0} = \{\, J_1^1 , J_2^1 \,\} \quad \textit{for } s = 0 \textit{ from Eq. (7.11)}$$
$$\texttt{Rel}_{3,10} = \{\, J_1^2 \,\} \quad \textit{for } s = 10 \textit{ from Eq. (7.11)}$$
$$\texttt{Rel}_{3,15} = \{\, J_2^2 \,\} \quad \textit{for } s = 15 \textit{ from Eq. (7.11)}$$
$$\texttt{Rel}_{3,20} = \{\, J_1^3 \,\} \quad \textit{for } s = 20 \textit{ from Eq. (7.11)}$$
$$\texttt{Rel}_{3,30} = \{\, J_1^4 , J_2^3 \,\} \quad \textit{for } s = 30 \textit{ from Eq. (7.11)}$$

Five vertically-composed tasks are formed since there are five time points in S_3 at each of which new higher-priority jobs are released. The five vertically-composed tasks are $V_{\{0\}}$, $V_{\{10\}}$, $V_{\{15\}}$, $V_{\{20\}}$ and $V_{\{30\}}$. For each vertically-composed task $V_{\{s\}}$, the value of `w(k,{s})` *for $k = 0, 1, 2$ is given in each row of Table 7.3 for $k = 0, 1, 2$ and $s = 0, 10, 15, 20, 30$.*

$V_{\{s\}}$	$k = 0$	$k = 1$	$k = 2$
$V_{\{0\}}$	`w(0,{0})=6`	`w(1,{0})=10`	`w(2,{0})=12`
$V_{\{10\}}$	`w(0,{10})=3`	`w(1,{10})=5`	`w(2,{10})=8`
$V_{\{15\}}$	`w(0,{15})=3`	`w(1,{15})=7`	`w(2,{15})=9`
$V_{\{20\}}$	`w(0,{20})=3`	`w(1,{20})=5`	`w(2,{20})=8`
$V_{\{30\}}$	`w(0,{30})=6`	`w(1,{30})=10`	`w(2,{30})=10`

Table 7.3: *The value of* `w(k,{s})` *for each $s \in S_3$ and for $k = 0, 1, 2$. The k faults affect the higher-priority jobs that are released at time $s \in S_3$.*

Run-time complexity for vertical composition: Calculating $\texttt{Rel}_{i,s}$ for all $s \in S_i$ needs total $O(\hat{N})$ operations. Calculating $L_k(\texttt{Rel}_{i,s})$ for set $\texttt{Rel}_{i,s}$ requires $O(|\texttt{Rel}_{i,s}| \cdot f^2)$ operations for all $k = 0, 1, 2, \ldots f$. There are at most $\hat{N}$ jobs that are released within any time interval of length D_{max}. Therefore, the number of total jobs having higher priority than the priority of task τ_i that are released in all the time points in set S_i is equal to $O(\hat{N})$. In other words, $\sum_{s \in S_i} |\texttt{Rel}_{i,s}| = O(\hat{N})$. Therefore, the computational complexity of all the vertical compositions in all time points $s \in S_i$ is $[O(\hat{N}) + O(\sum_{s \in S_i} |\texttt{Rel}_{i,s}| \cdot f^2)] = O(\hat{N} \cdot f^2)$. □

For each $s \in S_i$, a vertically-composed task $V_{\{s\}}$ is formed. The vertically-composed task $V_{\{s\}}$ has execution time `w(k,{s})` considering a k-fault pattern for $k = 0, 1, 2 \ldots f$. Within the interval $[0, D_i)$, there may be more than one vertically-composed task. In our running example, there are five vertically-composed task within $[0, D_3)$ as shown in Figure 7.2 for the schedulability analysis of task τ_3. The higher-priority jobs represented by two or more vertically-composed tasks will execute in $[0, D_i)$. Notice that the execution of the jobs represented by two or more vertically-composed tasks may not be completely independent. Some jobs in one vertically-composed task may interfere or be interfered by the execution of some jobs in another vertically-composed task within $[0, D_i)$. By considering such effect of one composed task over another, the vertically-composed tasks are further composed using horizontal composition to calculate `Load-Factor-HPi`.

Horizontal Composition

A horizontally-composed task is formed by composing two or more vertically-composed tasks. To see how this composition works, consider two different time points s_1 and s_2 in set S_i such that $s_1 < s_2$. For these two time points, two vertically-composed tasks $V_{\{s_1\}}$ and $V_{\{s_2\}}$ are formed during vertical composition. A horizontally-composed task, denoted by $H_{\{s_1, s_2\}}$, is formed by composing the two vertically composed tasks $V_{\{s_1\}}$ and $V_{\{s_2\}}$. The task $H_{\{s_1, s_2\}}$ abstracts all the jobs of the higher-priority tasks than the priority of task τ_i that are released at time instants s_1 and s_2.

The execution time of this new horizontally-composed task $H_{\{s_1, s_2\}}$ is denoted by `w(k,{`s_1, s_2`})` and must not be greater than $(D_i - s_1)$. This is because the earliest time at which the jobs represented by the the composed task $H_{\{s_1, s_2\}}$ can start execution is at time s_1 since $s_1 < s_2$. Note that, if $0 \in \{s_1, s_2\}$, then `w(k,{`s_1, s_2`})` must not be greater than D_i. The value of `w(k,{`s_1, s_2`})` represents the maximum execution exclusively by the jobs released at time s_1 and s_2 within the time interval $[0, D_i)$.

When considering the schedulability of task τ_i, there are a total of $|S_i|$ time instants at each of which a vertically-composed task is formed. To calculate the value of `Load-Factor-HPi`, one has to find the final horizontally-composed task H_{S_i} with execution time `w(k,`S_i`)` for all $k = 0, 1, 2 \ldots f$. The value of `w(k,`S_i`)` is the amount of execution completed by the higher-priority jobs that are released within the time instants in set S_i in $[0, D_i)$. Since set S_i contains all the time instants where jobs of higher-priority task are released, the value of `w(k,`S_i`)` is `Load-Factor-HPi`.

To find the horizontally-composed task H_{S_i}, total $(|S_i| - 1)$ horizontal compositions are needed. Starting with two vertically-composed tasks, a new horizontally-composed task is first formed. This horizontally-composed task is further composed with a third vertically-composed task to form the second horizontally-composed task. This process continues until all the vertically-composed tasks are considered in the horizontal compositions. Note that a vertically-composed task has no priority associated with it. The jobs (primary and backups) abstracted by a vertically-composed tasks have DM priorities. Therefore, the order of execution of the jobs abstracted by a horizontally-composed task is determined by the DM priorities of the jobs that are abstracted by the constituent vertically-composed tasks.

The first horizontally-composed task abstracts all higher-priority jobs released at two points that are in set S_i. The last (final) horizontally-composed task abstracts all the jobs that are released at all time points in set S_i. For example, the five vertically-composed tasks in Figure 7.2 are composed horizontally as shown in Figure 7.3.

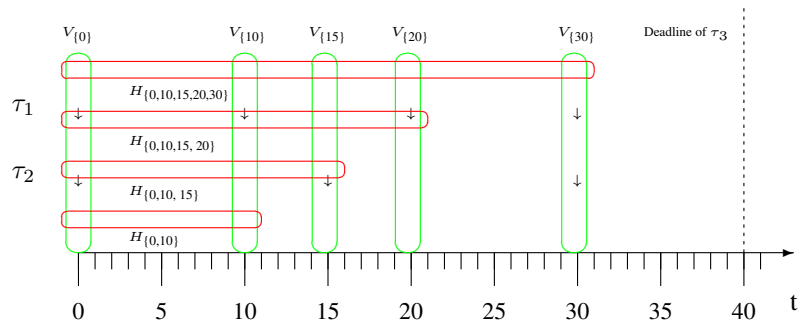

Figure 7.3: *Four horizontal compositions (horizontally longer ovals) are shown for the five vertically-composed tasks (vertically longer ovals). The four horizontally-composed tasks are $H_{\{0,10\}}$, $H_{\{0,10,15\}}$, $H_{\{0,10,15,20\}}$ and $H_{\{0,10,15,20,30\}}$. The execution time of $H_{\{0,10,15,20,30\}}$ is the value of* `Load-Factor-HPi`.

The technique to find the execution time of a horizontally-composed task is demonstrated next. If there are c time points in the set S_i, then the set S_i is represented as S_i=$\{s_1, s_2 \ldots s_c\}$ where $s_i < s_{i+1}$. According to Eq. (7.8), the set S_i contains the time point 0 and therefore, $s_1 = 0$. The first x time points in S_i is denoted by set

$$\texttt{p(x)} = \{s_l \mid l \leq x \text{ and } s_l \in S_i\}$$

Therefore, the set `p(x)` =$\{s_1, s_2 \ldots s_x\}$ for $x = 1, 2 \ldots c$. For example, we have `p(1)` =$\{s_1\}$=$\{0\}$, `p(2)` =$\{s_1, s_2\}$=$\{0, s_2\}$, and `p(c)` =$\{s_1, s_2 \ldots s_c\}$=S_i.

We start composing the first two vertically-composed tasks horizontally. The horizontal composition of the first two vertically-composed tasks $V_{\{s_1\}}$ and $V_{\{s_2\}}$ is denoted by the composed task $H_{\texttt{p(2)}}$=$H_{\{s_1, s_2\}}$. The execution time of $V_{\{s_1\}}$ and $V_{\{s_2\}}$ are `w(k,`$\{s_1\}$`)` and `w(k,`$\{s_2\}$`)`, respectively (can be computed using Eq.(7.12)). The

execution time of $H_{\texttt{p(2)}}$is denoted by $\texttt{w(k,p(2))} = \texttt{w(k,}\{s_1, s_2\}\texttt{)}$ and is given in Eq. (7.13) as follows, for $k = 0, 1, 2, \ldots f$:

$$\texttt{w(k,p(2))} = \max_{q=0}^{k} \left\{ min\Big\{ [\texttt{w(q,}\{s_1\}\texttt{)} + \texttt{w(k-q,}\{s_2\}\texttt{)}],\ D_i \Big\} \right\} \tag{7.13}$$

The calculation of the value of $\texttt{w(k,p(2))}$in Eq. (7.13) considers the sum of the execution time of tasks $V_{\{s_1\}}$ and $V_{\{s_2\}}$ considering respectively q and $(k-q)$ fault pattern such that the sum is maximized for some q, $0 \leq q \leq k$. Since the amount of execution within the interval $[0, D_i)$ by the higher-priority jobs released at time s_1 and s_2 can not be greater than $(D_i - s_1) = (D_i - 0) = D_i$, the minimum of this sum (for some q) and D_i is determined to be the value of $\texttt{w(k,p(2))}$ in Eq. (7.13). This is because the earliest time that higher-priority jobs can start execution is at time $s_1 = 0$.

By assuming that the value of $\texttt{w(k,p(x))}$ is known for the horizontally-composed tasks $H_{\texttt{p(x)}}$, a new horizontally-composed task $H_{\texttt{p(x+1)}}$=$H_{\texttt{p(x)}} \cup \{s_{x+1}\}$ is formed. The execution time $\texttt{w(k,p(x+1))}$ of the horizontally-composed task $H_{\texttt{p(x+1)}}$is given in Eq. (7.14), for $k = 0, 1, 2, \ldots f$, as follows:

$$\begin{aligned} &\texttt{w(k,p(x+1))} = \texttt{w(k,p(x)} \cup \{s_{x+1}\}\texttt{)} \\ &= \max_{q=0}^{k} \left\{ min\Big\{ [\texttt{w(q,p(x))} + \texttt{w(k-q,}\{s_{x+1}\}\texttt{)}],\ D_i \Big\} \right\} \end{aligned} \tag{7.14}$$

The execution time $\texttt{w(k,p(x+1))}$ of the new horizontally-composed task $H_{\texttt{p(x+1)}}$ is calculated by finding the sum of the execution time of the horizontally composed task $H_{\texttt{p(x)}}$ and the execution time of a new vertically-composed task $V_{\{s_{x+1}\}}$. The value of this sum is maximized by considering q fault-pattern in task $H_{\texttt{p(x)}}$ and $(k-q)$ fault-pattern in task $V_{\{s_{x+1}\}}$, for some q, $0 \leq q \leq k$. Since the amount of execution within the interval $[0, D_i)$ can not be greater than $(D_i - s_1) = (D_i - 0) = D_i$, the minimum of this sum (for some q) and D_i is the value of $\texttt{w(k,p(x+1))}$ in Eq. (7.14).

Using Eq. (7.14), the execution time $\texttt{w(k,}S_i\texttt{)}$ of the final horizontally-composed task H_{S_i}=$H_{\texttt{p(}|S_i|\texttt{)}}$ can be determined, for $k = 0, 1, 2 \ldots f$. The value of $\texttt{w(k,}S_i\texttt{)}$ is the value of `Load-Factor-HPi` for $k = 0, 1, 2 \ldots f$. Before the calculation of the execution time of horizontally-composed task is demonstrated using an example, the run-time complexity of horizontal composition is derived.

Run time complexity of horizontal compositions: There are total $(|S_i| - 1)$ horizontal composition for $|S_i|$ vertically-composed tasks when considering the schedulability analysis of task τ_i. When considering the schedulability of a task τ_i, for each horizontal composition, there are $(k+1)$ possibilities for q, $0 \leq q \leq k$, in Eq. (7.14). For each value of q, there is one addition and one comparison operation. Therefore, total $(2 \cdot (k+1))$ operations are needed for one horizontal composition for each k. For all $k = 0, 1, 2 \ldots f$, each horizontal composition requires total $[2+4+6+\ldots 2 \cdot (f+1)]$=$O(f^2)$ operations. Given all the $|S_i|$ vertical compositions, there are a total of $[(|S_i| - 1) \cdot O(f^2)]$= $O(|S_i| \cdot f^2)$ operations for all the $(|S_i| - 1)$ horizontal compositions. Note that $|S_i|$=$O(\hat{N})$ since there are at most $\hat{N}$ time instants where new higher-priority jobs are released. Therefore,

finding the `Load-Factor-HPi` for one task τ_i is $O(\hat{N} \cdot f^2)$. The time complexity to find the execution time of vertically-composed tasks is $O(\hat{N} \cdot f^2)$. Therefore, total time complexity for the vertical and horizontal composition when considering the schedulability of task τ_i is $O(\hat{N} \cdot f^2 + \hat{N} \cdot f^2)$=$O(\hat{N} \cdot f^2)$. □

Now the calculation of `Load-Factor-HPi` (that is, the value of `w(k,`S_i`)`) using our running example is presented.

Example 7.9. *For task τ_1, we have $S_1 = \emptyset$ from Eq.* (7.9)*. Therefore, no vertical composition, and hence no horizontal composition is needed.*

For task τ_2, we have $S_2 = \{0, 10\}$. Using vertical composition, we have two vertically-composed tasks $V_{\{0\}}$ and $V_{\{10\}}$. The execution time `w(k,{s})` *of the vertically-composed task for $s = 0$ and $k = 0, 1, 2$ fault patterns are* `w(0,{0})`*=3,* `w(1,{0})`*=5, and* `w(2,{0})`*=8 (given in the first column of Table 7.2 in page 131). Similarly, the execution time* `w(k,{s})` *of the vertically-composed task for $s = 10$ and $k = 0, 1, 2$ fault patterns are determined as* `w(0,{10})`*=3,* `w(1,{10})`*=5 and* `w(2,{10})`*=5 (given in the second column of Table 7.2 in page 131).*

The two vertically-composed tasks $V_{\{0\}}$ and $V_{\{10\}}$ are horizontally-composed as $H_{\{0,\ 10\}}$ and its execution time `w(k,{0,10})` *using Eq.*(7.13) *is calculated in Table 7.4 for $k = 0, 1, 2$. Form Table 7.4, when considering the schedulability of task τ_2, the amount of execution completed by the higher-priority jobs within $[0, 15)$ is 6, 8 and 11 for k=0, 1 and 2 errors affecting only the jobs of the higher-priority task, respectively.*

For task τ_3, we have $S_3 = \{0, 10, 15, 20, 30\}$. Using vertical composition, we have five vertically-composed tasks $V_{\{0\}}$, $V_{\{10\}}$, $V_{\{15\}}$, $V_{\{20\}}$ and $V_{\{30\}}$. The execution time of the vertically-composed tasks for $k = 0, 1, 2$ are given in Table 7.3. Using Eq. (7.13) *and Eq.* (7.14)*, the execution time of the four horizontally composed tasks formed using the five vertically-composed tasks $V_{\{0\}}$, $V_{\{10\}}$,$V_{\{15\}}$ $V_{\{20\}}$ and $V_{\{30\}}$ is calculated. The execution time of the horizontally-composed task $H_{\{0,\ 10,\ 15,\ 20,\ 30\}}$ is* `w(k,{0,10,15,20,30})` *that is calculated using Eq.* (7.14)*, for $k = 0, 1, 2$ (given in the fourth row of each Table 7.5-Table 7.7).*

By composing $V_{\{0\}}$ and $V_{\{10\}}$ horizontally, the new horizontally-composed task is $H_{\{0,10\}}$ is formed using Eq. (7.13)*. The execution time of the horizontally-composed task $H_{\{0,10\}}$ is* `w(k,{0,10})` *and calculated using Eq.* (7.13) *for $k = 0, 1, 2$ (given in the first row of each Table 7.5-Table 7.7).*

Then, the first horizontally-composed task $H_{\{0,\ 10\}}$ and the vertically-composed task $V_{\{15\}}$ are composed to form the second horizontally-composed task $H_{\{0,\ 10,\ 15\}}$. The execution time of $H_{\{0,10,15\}}$ is `w(k,{0,10,15})` *and determined using Eq.* (7.14) *for $k = 0, 1, 2$ (given in the second row of each Table 7.5-Table 7.7). This process continues and finally the horizontally-composed task $H_{\{0,\ 10,\ 15,\ 20\}}$ and the vertically-composed task $V_{\{30\}}$ are composed into the final horizontally-composed task that is $H_{\{0,\ 10,\ 15,\ 20,\ 30\}}$. The execution time of the four horizontally-composed tasks are given in Table 7.5, Table 7.6 and Table 7.7 for $k = 0$, $k = 1$ and $k = 2$ fault patterns, respectively.*

The amount of execution time `w(k,{0,10,15,20,30})` *of the final horizontally-composed task H_{S_i} is the exact value of* `Load-Factor-HPi` *due to a k-fault-pattern.*

For $H_{\{0,\,10\}}$ and $k = 0$

$$\texttt{w(0,\{0,10\})} = \texttt{w(0,\{0\}}\cup\texttt{\{10\})}$$
$$= \max_{q=0}^{0} \Big\{ min\{\texttt{w(q,\{0\})} + \texttt{w(k-q,\{10\})}, D_i\} \Big\}$$
$$= min\Big\{ [\texttt{w(0,\{0\})} + \texttt{w(0,\{10\})}], D_i \Big\}$$
$$= min\{[3+3], 15\} = min\{6, 15\} \Big\} = 6$$

For $H_{\{0,10\}}$ and $k = 1$

$$\texttt{w(1,\{0,10\})} = \texttt{w(1,\{0\}}\cup\texttt{\{10\})}$$
$$= \max_{q=0}^{1} \Big\{ min\{\texttt{w(q,\{0\})} + \texttt{w(1-q,\{10\})}, D_i\} \Big\}$$
$$= max\Big\{ min\{[\texttt{w(0,\{0\})} + \texttt{w(1,\{10\})}], D_i\},$$
$$min\Big\{ [\texttt{w(1,\{0\})} + \texttt{w(0,\{10\})}], D_i\} \Big\}$$
$$= max\Big\{ min\{[3+5], 15\}, min\{[5+3], 15\} \Big\}$$
$$= max\Big\{ min\{8, 15\}, min\{8, 15\} \Big\} = 8$$

For $H_{\{0,10\}}$ and $k = 2$

$$\texttt{w(2,\{0,10\})} = \texttt{w(2,\{0\}}\cup\texttt{\{10\})}$$
$$= \max_{q=0}^{2} \Big\{ min\{\texttt{w(q,\{0\})} + \texttt{w(2-q,\{10\})}, D_i\} \Big\}$$
$$= max\Big\{ min\{[\texttt{w(0,\{0\})} + \texttt{w(2,\{10\})}], D_i\},$$
$$min\{[\texttt{w(1,\{0\})} + \texttt{w(1,\{10\})}], D_i\}$$
$$min\{[\texttt{w(2,\{0\})} + \texttt{w(0,\{10\})}], D_i\} \Big\}$$
$$= max\Big\{ min\{[3+5], 15\}, min\{[5+5], 15\}, min\{[8+3], 15\} \Big\}$$
$$= max\Big\{ min\{8, 15\}, min\{10, 15\}, min\{11, 15\} \Big\} = 11$$

Table 7.4: *Calculation of* `w(k,{0,10})` *for horizontally-composed task* $H_{\{0,\,10\}}$ *for* $k = 0, 1, 2$.

Composed task	Execution time for 0-fault pattern
$H_{\{0,\,10\}}$	`w(0,{0,10})`=9
$H_{\{0,10,15\}}$	`w(0,{0,10,15})`=12
$H_{\{0,10,15,20\}}$	`w(0,{0,10,15,20})`=15
$H_{\{0,10,15,20,30\}}$	`w(0,{0,10,15,20,30})`=21

Table 7.5: *The execution time due to 0-fault pattern of the four horizontally-composed tasks* $H_{\{0,\,10\}}$, $H_{\{0,\,10,\,15\}}$, $H_{\{0,\,10,\,15,\,20\}}$, *and* $H_{\{0,\,10,\,15,\,20,\,30\}}$

The value of `w(k,{0,10,15,20,30})` *represents the amount of execution time within* $[0, 40)$ *by all the higher-priority jobs due to the* k*-fault-pattern. Table 7.5-*

Composed task	Execution time for 1-fault pattern
$H_{\{0,\ 10\}}$	`w(1,{0,10})`=13
$H_{\{0,10,15\}}$	`w(1,{0,10,15})`=16
$H_{\{0,10,15,20\}}$	`w(1,{0,10,15,20})`=19
$H_{\{0,10,15,20,30\}}$	`w(1,{0,10,15,20,30})`=25

Table 7.6: *The execution time due to 1-fault pattern of the four horizontally-composed tasks $H_{\{0,\ 10\}}$, $H_{\{0,\ 10,\ 15\ \}}$, $H_{\{0,\ 10,\ 15,\ 20\}}$, and $H_{\{0,\ 10,\ 15,\ 20,\ 30\}}$*

Composed task	Execution time for 2-fault pattern
$H_{\{0,\ 10\}}$	`w(2,{0,10})`=18
$H_{\{0,10,15\}}$	`w(2,{0,10,15})`=21
$H_{\{0,10,15,20\}}$	`w(2,{0,10,15,20})`=24
$H_{\{0,10,15,20,30\}}$	`w(2,{0,10,15,20,30})`=30

Table 7.7: *The execution time due to 2-fault pattern of the four horizontally-composed tasks $H_{\{0,\ 10\}}$, $H_{\{0,\ 10,\ 15\ \}}$, $H_{\{0,\ 10,\ 15,\ 20\}}$, and $H_{\{0,\ 10,\ 15,\ 20,\ 30\}}$*

Table 7.7 show that the execution completed by the higher-priority jobs within $[0, 40)$ is 21, 25, and 30 for k=0,1 and 2-fault patterns, respectively (shown in the shaded fourth row in each of the Table 7.5-Table 7.7). □

It is easy to realize at this point that the way the composition technique is applied to calculate the execution time of the final horizontally composed task can also be applied to any fixed-priority task system and to any length of the interval rather than $[0, D_i)$. Based on the value of the `Load-Factor-HPi`, the exact `FTDM` schedulability condition of task τ_i is derived in Section 7.6.

7.6 Exact Schedulability Test

The exact schedulability condition for `FTDM` scheduling of a sporadic task set Γ is derived based on the exact schedulability condition of each task τ_i for $i = 1, 2 \ldots n$. The exact schedulability condition of task τ_i depends on the amount of execution required by task τ_i and its higher-priority jobs within the interval $[0, D_i)$ considering at most f errors that could occur within $[0, D_i)$.

By considering $(f - k)$ faults exclusively affecting task τ_i and the k-fault pattern affecting the higher-priority jobs of task τ_i within the interval $[0, D_i)$, the sum of `Load-Factor-i` and `Load-Factor-HPi` can be calculated such that it is maximized for some k, $0 \leq k \leq f$. This sum is consequently the worst-case workload within $[0, D_i)$. The value of `Load-Factor-i` is $C_i^{(f-k)}$ and can be calculated using Eq. (7.2), for $k = 0, 1, 2, \ldots f$. The value of `Load-Factor-HPi` is `w(k,`S_i`)` and can be calculated using Eq. (7.14), for $k = 0, 1, 2, \ldots f$.

The maximum total workload within $[0, D_i)$ is denoted by $\texttt{TLoad}_i$ which is equal to the sum of `Load-Factor-i` and `Load-Factor-HPi` such that this sum is maximum for some k, $0 \leq k \leq f$. The function $\texttt{TLoad}_i$ is defined in Eq. (7.15):

$$\texttt{TLoad}_i \;=\; \max_{k=0}^{f} \{\, C_i^{(f\,-\,k)} \;+\texttt{w(k,}S_i\texttt{)}\} \tag{7.15}$$

Using Eq. (7.15), the maximum total workload within the interval $[0, D_i)$ can be determined. The total load is equal to the sum of the execution time required by task τ_i if $(f - k)$ errors exclusively affect the task τ_i and the execution time within the interval $[0, D_i)$ by the jobs having higher priority than the task τ_i due to k-fault pattern, such that, the sum is maximum for some k, $0 \leq k \leq f$.

Run-time complexity to compute the total load: Calculating the value of $C_i^{(f\,-\,k)}$ for all $k = 0, 1, 2, \ldots f$ can be done in $O(f)$ steps. The value of $\texttt{w(k,}S_i\texttt{)}$ is the execution time of the final horizontally-composed task and can be calculate in $O(\hat{N} \cdot f^2)$ time for all $k = 0, 1, 2, \ldots f$. In Eq. (7.15), there are $(f + 1)$ possible values for the selection of k, $0 \leq k \leq f$. Evaluating $\texttt{TLoad}_i$ in Eq. (7.15) requires a total of $(f + 1)$ addition operations and f comparisons to find the maximum. Given the values of $C_i^{(f\,-\,k)}$ and $\texttt{w(k,}S_i\texttt{)}$ for all $k = 0, 1, 2, \ldots f$, finding the value of $\texttt{TLoad}_i$ requires $O(f)$ steps. Therefore, the total time complexity for evaluating $\texttt{TLoad}_i$ is $[O(f)+O(\hat{N} \cdot f^2)+O(f)]=O(\hat{N} \cdot f^2)$. □

Based on the value of $\texttt{TLoad}_i$, the necessary and sufficient schedulability condition of task τ_i in `FTDM` scheduling is proposed in Theorem 7.1.

Theorem 7.1. *Task $\tau_i \in \Gamma$ is `FTDM`-schedulable if and only if $\texttt{TLoad}_i \;\leq\; D_i$.*

Proof. (**if part**) It will be shown using proof by contradiction that if $\texttt{TLoad}_i \leq D_i$, then task τ_i is `FTDM`-schedulable. The value of $\texttt{TLoad}_i$ is the sum of two workload factors: `Load-Factor-i` and `Load-Factor-HPi`. The value of `Load-Factor-i` is the maximum execution time required by the task τ_i if $(f-k)$ errors exclusively occur in the first job of task τ_i. The value of `Load-Factor-i` is given by $C_i^{(f\,-\,k)}$ in Eq. (7.2) for $k = 0, 1, 2, \ldots f$. The value of `Load-Factor-HPi` is the execution completed within the interval $[0, D_i)$ by the jobs having higher priority than the priority of task τ_i. The value of `Load-Factor-HPi` is given by $\texttt{w(k,}S_i\texttt{)}$ which is equal to the execution time of the final horizontally-composed task H_{S_i} considering a k-fault pattern affecting the jobs of the higher-priority tasks within the interval $[0, D_i)$, for $k = 0, 1, 2, \ldots f$. The value of $\texttt{w(k,}S_i\texttt{)}$ is the maximum amount of work that can be completed by the higher-priority jobs within $[0, D_i)$.

Now, assume a contradiction, that is, that some job of task τ_i misses it deadline when $\texttt{TLoad}_i \leq D_i$. This assumption implies that the first job of task τ_i misses its deadline (due to the first job being released at a critical instant). When the first job of task τ_i misses its deadline at time D_i, the processor must be continuously busy within the entire interval $[0, D_i)$. This is because, if the processor was idle at some time instant

within $[0, T_i)$, then τ_i could not have missed its deadline since FTDM scheduling is based on work-conserving DM scheduling.

In case that τ_i misses its deadline, the processor either executes task τ_i or its higher-priority jobs at each time instant within $[0, D_i)$. The time required for executing the higher-priority jobs within $[0, D_i)$ is Load-Factor-HPi which is given by w(k, S_i). Note that w(k, S_i) is less than or equal to D_i (because of the *min* function) according to Eq. (7.14). The total time required for completing the execution of task τ_i is Load-Factor-i considering $(f - k)$ errors that could affect the first job of task τ_i. Since τ_i misses it deadline at D_i, the complete execution of task τ_i can not have finished by time D_i. Therefore, the sum of Load-Factor-i and Load-Factor-HPi, denoted by TLoad$_i$, must have been greater than D_i (which is a contradiction!). Therefore, if TLoad$_i \leq T_i$, then task τ_i is FTDM-schedulable.

(**only if part**) It will be shown that, if τ_i is FTDM-schedulable, then TLoad$_i \leq T_i$. The amount of work on behalf of task τ_i (including execution of its backup) completed in the FTDM schedule in $[0, D_i)$ is Load-Factor-i. Since when analyzing the schedulability of task τ_i, the amount of execution on behalf of the jobs (including execution of their backups) having higher priority than task τ_i that is completed by FTDM scheduling is exactly equal to Load-Factor-HPi within $[0, D_i)$.

Since the work completed by algorithm FTDM on behalf of the jobs in (HPJ$_i \cup \{J_i^1\}$) in $[0, D_i)$ is equal to the sum of Load-Factor-i and Load-Factor-HPi, the total load TLoad$_i$ is less than or equal to D_i whenever task τ_i is FTDM schedulable. Therefore, if task τ_i is fault-tolerant FTDM-schedulable, then TLoad$_i \leq D_i$. □

The exact schedulability test for FTDM scheduling of task τ_i is given in Theorem 7.1. The time complexity for evaluating the exact test is same as the time complexity for evaluating Eq. (7.15). Therefore, the necessary and sufficient condition for checking the schedulability of task τ_i can be evaluated in time $O(\hat{N} \cdot f^2)$. The exact schedulability condition for the entire task set Γ is now given in the following Corollary 7.1.

Corollary 7.1. *Task set* $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ *is FTDM-schedulable if, and only if, task* τ_i *is FTDM-schedulable using Theorem 7.1 for all* $i = 1, 2, \ldots n$.

Note that Corollary 7.1 is the application of Theorem 7.1 for each one of the n tasks in set Γ. Therefore, the exact schedulability condition for the entire task set can be evaluated in $O(n \cdot \hat{N} \cdot f^2)$ time. The FTDM-schedulability of the running example task set given in Table 7.1 is now demonstrated.

Example 7.10. *We have to apply Theorem 7.1 to all the three tasks given in Table 7.1. For task* τ_i*, the value of* TLoad$_i$ *for* $i = 1, 2, 3$ *has to be computed. The task* τ_1 *being the highest priority task is trivially FTDM-schedulable.*

Consider the schedulability of task τ_2*. Remember that,* w(k, S_i) *is the execution time of the final horizontally-composed task and is equal to* Load-Factor-HPi*. For task* τ_2*, we have* $S_2 = \{0, 10\}$*. By horizontal composition, the final horizontally-composed task* $H_{\{0,10\}}$ *has execution time equal to* w(0, S_2) = 6*,* w(1, S_2) = 8*, and* w(2, S_2) = 11 *for* $k = 0$*,* $k = 1$ *and* $k = 2$ *fault-patters within interval* $[0, 15)$

(given in Table 7.4), respectively. For task τ_2, we also have C_2^0 =3, C_2^1 =7 and C_2^2 =9 for $k = 0$, $k = 1$ and $k = 2$ fault-patterns, respectively, which are the values of `Load-Factor-i` *using Eq.*(7.3)*. For task τ_2 and $f = 2$, the calculation of* $\texttt{TLoad}_2$ *using Eq.* (7.15) *is given below:*

$$\begin{aligned}\texttt{TLoad}_2 &= \max_{q=0}^{2}\Big\{C_2^{(2-q)} + \texttt{w(q,\{0,10\})}\Big\}\\ &= max\Big\{[C_2^2 + \texttt{w(0,\{0,10\})}], [C_2^1 + \texttt{w(1,\{0,10\})}],\\ &\qquad [C_2^0 + \texttt{w(2,\{0,10\})}]\Big\} = max\Big\{[9+6],[7+8],[3+11]\Big\} = 15\end{aligned}$$

Since $\texttt{TLoad}_2 = 15 \leq D_2 = 15$*, task τ_2 is* `FTDM`*-schedulable using Theorem 7.1.*

Consider the schedulability of task τ_3. We have $S_3 = \{0, 10, 15, 20, 30\}$. By horizontal composition, the final horizontally-composed task $H_{\{0,10,15,20,30\}}$ has execution time equal to $\texttt{w(0,}S_3\texttt{)}$*=21,* $\texttt{w(1,}S_3\texttt{)}$*=25, and* $\texttt{w(2,}S_3\texttt{)}$*=30 for $k = 0$, $k = 1$ and $k = 2$ fault-patterns, within interval $[0, 40)$ (given in the fourth shaded row in Table 7.5–Table 7.7), respectively. For task τ_3, we also have C_3^0 =9, C_3^1 =17 and C_3^2 =23 for $k = 0$, $k = 1$ and $k = 2$ fault-patterns, respectively, which are the values of* `Load-Factor-i` *using Eq.*(7.3)*. For task τ_3 and $f = 2$, the calculation of* $\texttt{TLoad}_3$ *using Eq.* (7.15) *is given below:*

$$\begin{aligned}\texttt{TLoad}_3 &= \max_{q=0}^{2}\Big\{C_3^{(2-q)} + \texttt{w(q,\{0,10,15,20,30\})}\Big\}\\ &= max\Big\{[C_3^2 + \texttt{w(0,\{0,10,15,20,30\})}],\\ &\qquad [C_3^1 + \texttt{w(1,\{0,10,15,20,30\})}],\\ &\qquad [C_3^0 + \texttt{w(2,\{0,10,15,20,30\})}],\Big\}\\ &= max\Big\{[21+23],[25+17],[30+9]\Big\} = 44\end{aligned}$$

Since $\texttt{TLoad}_3 = 44 \geq D_3 = 40$*, task τ_3 is not* `FTDM`*-schedulable using Theorem 7.1. Therefore, the task set given in Table 7.1 is not* `FTDM`*-schedulable using Corollary 7.1.*

Based on the necessary and sufficient schedulability condition in Corollary 7.1, the pseudocode of the schedulability test for `FTDM` scheduling is now algorithmically presented in Section 7.7.

7.7 Algorithm for the `FTDM` Schedulability Test

In this section, the exact test for fault-tolerant scheduling algorithm `FTDM` based on the exact schedulability condition derived in Corollary 7.1 is presented. First, the pseudocode of the algorithm `CheckFeasibility(`τ_i, f`)`is given in Figure 7.4. The algorithm `CheckFeasibility(`τ_i, f`)`checks the `FTDM` schedulability of a task τ_i by

considering occurrences of f task errors in any jobs of the tasks in set $\{ \tau_1, \tau_2, \ldots \tau_i\}$ released within the interval $[0, D_i)$. Next, the algorithm `FTDM-Test`(Γ, f)that checks the schedulability of the entire task set Γ based on the schedulability of each task $\tau_i \in \Gamma$ is presented in Figure 7.5.

Algorithm `CheckFeasibility(`τ_i, f`)`

1. Find the $\mathtt{HPJ}_i$ using Eq. (7.4)
2. Find the S_i using Eq. (7.8)
3. **For all** $s \in S_i$
4. **For** $k = 0$ **to** f
5. Find `w(k,{`s`})` using Eq. (7.12)
6. **End For**
7. **End For**
8. **For** $x = 2$ **to** $|S_i|$
9. **For** $k = 0$ **to** f
10. Find `w(k,p(`x`-1)`$\cup$`{`s_x`})` using Eq. (7.14)
11. **End For**
12. **End For**
13. **For** $k = f$ **to** 0
14. Find $C_i^{(f-k)}$ using Eq. (7.2)
15. **End For**
16. **For** $k = 0$ **to** f
17. **If** $[C_i^{(f-k)}$ `+w(k,`S_i`)]` $> D_i$ **then**
18. **return False**
19. **End If**
20. **End For**
21. **return True**

Figure 7.4: *Pseudocode of Algorithm* `CheckFeasibility(`τ_i, f`)`

In line 1 of Algorithm `CheckFeasibility(`τ_i, f`)`in Figure 7.4, the jobs having higher priority than the priority of task τ_i are determined using Eq. (7.4). In line 2, the time instants at each of which higher-priority jobs are released within the interval $[0, D_i)$ are determined using Eq. (7.8). Using the loop in line 3–7, the execution time `w(k,{`s`})` of each vertically-composed task $V_{\{s\}}$ is derived for each point $s \in S_i$. The value of `w(k,{`s`})` is determined for each $k = 0, 1, 2, \ldots f$ at line 5 using Eq. (7.12).

Using the loop in line 8–12, the vertically-composed tasks are composed further using horizontal compositions. The loop at line 8 iterates total $(|S_i| - 1)$ times. Each iteration of this loop calculates the execution time of one horizontally composed task $H_{\mathtt{p}(x)}$=$H_{\mathtt{p}(x\text{-}1)\cup\{s_x\}}$, for $x = 2, 3, \ldots |S_i|$. The execution time `w(k,p(`x`-1)`$\cup$`{`s_x`})` of the horizontally-composed task $H_{\mathtt{p}(x\text{-}1)\cup\{s_x\}}$ is calculated at line 10 using Eq. (7.14) for a k-fault pattern, $k = 0, 1, 2, \ldots f$. The execution time `w(k,`S_i`)` of the final

horizontally-composed task H_{S_i} is the value of `Load-Factor-HPi`, for $k = 0, \ldots f$.

Using the loop in line 13–15, the value of $C_i^{(f-k)}$ is determined in line 14 using Eq. (7.2) for $k = 0, 1, \ldots f$. Remember that the value of $C_i^{(f-k)}$ is `Load-Factor-i`. In line 16–20, the exact schedulability condition for τ_i is checked by considering k errors affecting the jobs of the higher-priority tasks and $(f-k)$ errors exclusively affecting the task τ_i, for $k = 0, 1, 2, \ldots f$. In line 17, the value of $\texttt{TLoad}_i$ is calculated by summing `Load-Factor-i` and `Load-Factor-HPi` and this sum is compared against the relative deadline of task τ_i. If this sum is greater than D_i, then task τ_i is not FTDM schedulable and the algorithm `CheckFeasibility(`τ_i, f`)`returns False at line 18. If the condition at line 17 is false for all $k = 0, 1, 2 \ldots f$, then task τ_i is FTDM-schedulable and the algorithm `CheckFeasibility(`τ_i, f`)`returns True at line 21. Next, using the algorithm `CheckFeasibility(`τ_i, f`)`the algorithm `FTDM-Test`$(\Gamma, f)$is presented in Figure 7.5.

Algorithm **`FTDM-Test`**$(\Gamma,$ $f)$

1. **For all** $\tau_i \in \{\tau_1, \tau_2, \ldots, \tau_n\}$
2. **If** `CheckFeasibility(`τ_i, f`)`**= False then**
3. **return False**
4. **End If**
5. **End For**
6. **return True**

Figure 7.5: *Pseudocode of Algorithm* `FTDM-Test`$(\Gamma,$ $f)$

Using the loop in line 1–5 of algorithm `FTDM-Test`(Γ, f) given in Figure 7.5, the FTDM-schedulability of task τ_i is checked. The algorithm `FTDM-Test`(Γ, f), based on algorithm `CheckFeasibility(`τ_i, f`)`, checks the FTDM schedulability of task $\tau_i \in \Gamma$ at line 2. If the condition at line 2 is true for any task τ_i (the algorithm `CheckFeasibility(`τ_i, f`)`returns False), then the task set Γ is not FTDM-schedulable. In such case, the algorithm `FTDM-Test`(Γ, f) returns False (line 3). If the condition at line 2 is false for task τ_i, for all $i = 1, 2, \ldots n$ (`CheckFeasibility(`τ_i, f`)`returns True for each task), then the task set Γ is FTDM-schedulable. In such case, the algorithm `FTDM-Test`(Γ, f) returns True (line 6). Given a task set Γ and the number of task errors f that can occur within any possible interval of length D_{max}, the fault-tolerant schedulability of the task set using the FTDM algorithm can be exactly determined using algorithm `FTDM-Test`(Γ, f) in $O(n \cdot \hat{N} \cdot f^2)$ time. The applicability of exact uniprocessor schedulability test for FTDM scheduling to multiprocessor platform is presented in subsection 7.7.1.

7.7.1 Multiprocessor Scheduling

The uniprocessor FTDM schedulability analysis is applicable to multiprocessor partitioned scheduling. The exact test of FTDM scheduling can be applied during the task assignment phase of a partitioned multiprocessor scheduling algorithm in which the run time dispatcher in each processor executes tasks in DM priority order using uniprocessor FTDM scheduling.

Consider a multiprocessor platform consisting of m identical processors. The question addressed is as follows:

> Is there an assignment of the tasks of set Γ on m processors such that each processor can tolerate f task errors within a time interval equal to the maximum relative deadline of the tasks assigned to each processor?

Partitioned multiprocessor task scheduling is typically based on a bin-packing algorithm for task assignment to the processors. When assigning a new task to a processor, a uniprocessor schedulability condition is used to check whether or not an unassigned task and all the previously assigned tasks in a particular processor are schedulable using uniprocessor scheduling, for example, DM scheduling algorithm. If the answer is yes, the unassigned task can be assigned to the processor. In order to extend the partitioned multiprocessor scheduling to fault-tolerant scheduling, we can apply the exact schedulability condition derived in Corollary 7.1 when trying to assign a new task to a processor in partitioned scheduling. The following principle discusses how the exact schedulability condition derived in Corollary 7.1 can be applied to the First-Fit heuristic for task assignment on multiprocessors.

An idea to assign tasks to multiprocessors: Consider the First-Fit heuristic for task assignment to processors. Given a task set $\{\tau_1, \tau_2, \ldots, \tau_n\}$, the tasks are to be assigned to m processors in increasing order of (given) task index. That is, task τ_1 is considered first, then task τ_2 is considered, and so on. Using the First-Fit heuristics, the processors of the multiprocessor platform are also indexed from $1 \ldots m$. An unassigned task is considered to be assigned to processor in increasing order of processor index. An unassigned task is assigned to the processor with the smallest index on which it is schedulable.

Following the First-Fit heuristic, task τ_1 is trivially assigned to the first processor. For task τ_2, the necessary and sufficient schedulability condition in Corollary 7.1 is applied to a set of tasks $\{\tau_1, \tau_2\}$ considering at most f errors that could occur in an interval of length D_{max} (where D_{max} is the maximum relative deadline of the tasks in set $\{\tau_1, \tau_2\}$). If the schedulability condition is satisfied, then τ_2 is assigned to the first processor. Otherwise, τ_2 is trivially assigned to the second processor. Similarly, for each unassigned task τ_i, the schedulability condition in Corollary 7.1 is first checked considering the already assigned tasks including task τ_i on the processor with index 1. If task τ_i and all the previously assigned tasks to the first processor are FTDM schedulable using the exact condition in Corollary 7.1, then τ_i is assigned to the first processor. If the exact condition is not satisfied, the schedulability condition is checked for the second processor and so on.

If task τ_i can not be assigned to any processor, then task set Γ can not be partitioned on the given multiprocessor platform. If all the tasks are assigned to the multiprocessor platform, then task set Γ is FTDM schedulable on each processor. For a successful partition of the task set Γ, each processor can tolerate f errors that can occur in any tasks within a time interval equal to the maximum relative deadline of the tasks assigned to each particular processor. The successful assignment of the tasks to m processors also guarantees that total $(m \cdot f)$ task errors (each processor tolerating at most f errors) can be tolerated within each of all possible time intervals of length D_{max} where D_{max} is the largest relative deadline of all the tasks.

7.8 Summary

This chapter presents the analysis of FTDM scheduling algorithm that can be used to guarantee the correctness and timeliness property of real-time applications on uniprocessor. The correctness property of the system is addressed by means of fault-tolerance so that the system functions correctly even in the presence of faults. The timeliness property is addressed by deriving a necessary and sufficient schedulability condition for the FTDM scheduling algorithm on uniprocessor.

The proposed algorithm FTDM-Test(Γ, f) can verify the FTDM-schedulability of constrained-deadline sporadic task sets. The time complexity to evaluate the test is $O(n \cdot \hat{N} \cdot f^2)$, where n is the number of tasks in the periodic task set, $\hat{N}$ is the maximum number of jobs released within any time interval of length D_{max}, and f is the maximum number of task errors that can occur within any time interval of length D_{max}.

The fault model considered for the FTDM schedulability analysis is general enough in the sense that multiple task errors due to various hardware and software faults can occur in any job, at any time and even during the recovery operation. There is no restriction posed on the inter-arrival time between the occurrences of any two consecutive faults. The only restriction of the fault model is that a maximum of f task errors could occur within any time interval of length D_{max}. Such a fault model does not require to know the distribution of the faults and also covers faults where they may arrive in bursts.

No other work has proposed an exact fault-tolerant schedulability analysis of sporadic tasks having constrained deadlines considering such a general fault model as is used in this chapter. If an efficient (in terms of time complexity) and exact schedulability test is needed, then the scheduling algorithm FTDM provides better computational efficiency than that of proposed for fault-tolerant EDF scheduling algorithm in [Ayd07]. The proposed exact uniprocessor schedulability condition can be applied to task scheduling on multiprocessors based on partitioned approach.

8 Fault-Tolerant Scheduling on Multiprocessors

In this chapter, a fixed-priority multiprocessor scheduling algorithm, called Fault-Tolerant Global Scheduling (FTGS), is proposed for tolerating both task errors and processor failures. The major strength of FTGS algorithm is the fault model it assumes; a variety of software and hardware faults that may lead to task errors or processor failures are considered. The main contribution is the derivation of a sufficient schedulability test for the proposed FTGS algorithm that exploits time redundancy to tolerate faults. This schedulability test when satisfied guarantees that all the deadlines of the real-time tasks are met even in the presence of task errors and processor failures.

The novelty of the proposed schedulability test is that the resilience of resource-constrained embedded real-time systems can be determined for different combinations of task errors and processor failures. The schedulability test for the FTGS algorithm is OPA-compatible: if a task set does not satisfy the schedulability test for a given priority ordering of the tasks, then a priority ordering for which the taskset may satisfy the schedulability test can be searched using multiprocessor extension of Audsley's optimal priority assignment algorithm.

8.1 Introduction

There are numerous works that have addressed fault-tolerance for partitioned and global scheduling on multiprocessors [OS94, GMM94, TKK95, BMR99, CYKT07, KLR10, BGJ06, LLMM99]. In fault-tolerant scheduling, each task is considered to have one pri-

mary and one or more backups. In partitioned fault-tolerant scheduling, a task allocation algorithm assigns the primary and backups of each task to distinct processors at design time. In case of a task error or processor failure detected at run time, the backup of the affected task is executed on a different non-faulty processor to which it is assigned.

One interesting observation of the task allocation algorithms proposed in [OS94, GMM94, TKK95, BMR99, CYKT07, KLR10] is that these algorithms do not take into account any difference between task errors and processor failures when assigning the tasks to the processors. These allocation algorithms pessimistically assume that tolerating a task error is equivalent to tolerating a processor failure. This pessimism requires relatively higher number of processors for successfully assigning all the primary and backups even when only task errors are to be tolerated. Such over provisioning of computing resources (i.e., processors) may restrict the use of partitioned method for many resource-constrained embedded real-time systems like automotive and avionics where weight, volume and space are limited. And more importantly, increasing the number of processors also increases the probability of having more faults in the system.

One of the consequences of the rising trend of transient faults in computer electronics (as pointed out by Baumann in [Bau05]) is the possibility of having higher number of task errors. It is therefore important to develop resource efficient fault-tolerant scheduling algorithm to tolerate task errors. Global scheduling algorithm does not require allocation of tasks to processors. The main motivation of the work in this chapter is based on an important observation: the global scheduler can simply dispatch the backup of the faulty task to any processor even to the processor on which the task has encountered a task error. This is because transient errors are short lived and tolerating such errors using global scheduling does not need the backups to be executed on different processors.

In this chapter, a **F**ault-**T**olerant **G**lobal **S**cheduling algorithm, called FTGS, to tolerate both task errors and processor failures is proposed. The algorithm FTGS not only can tolerate task errors but also can withstand processor failures. Global scheduling can tolerate processor failure just by assuming the task executing on the faulty processor has encountered an error. In other words, a processor failure can be viewed from the global scheduler's point of view as a task error. The treatment to tolerate the processor failure using FTGS algorithm is same as tolerating a task error — dispatching the primary and the backups of the tasks only to the non-faulty processors. By tolerating processor failure it means that the effect of permanent processor failure is mitigated by executing the tasks on non-faulty processors while meeting all the deadlines of the tasks.

Time-redundancy is considered to tolerate both task errors and processor failures. In order to ensure that all the deadlines of the application tasks are met while achieving fault-tolerance, the schedulability analysis of FTGS algorithm derives a sufficient schedulability test that when satisfied for a task set guarantees that all the deadlines are met. One of the novel properties of the proposed schedulability test is that the number of task errors and the number of processor failures are *separately* incorporated in to the mathematical expression of the schedulability test. This property enables the system designer to independently judge the robustness of the schedule in terms of tolerating only task errors, only processor failures, or tolerating both.

Another important property of the schedulability test for the FTGS scheduling algorithm is that it is OPA-compatible. If the proposed schedulability test is not satisfied for a task set for a given priority ordering, then another priority assignment for which the task set may satisfy the schedulability test can be determined. This an important property since the optimal priority assignment for global FP scheduling is not known.

The FTGS scheduling and its corresponding schedulability test consider a very general fault model in the sense that, *multiple errors* can occur in any task, at any time, in any processor, and even during the recovery operations. In many other works regarding fault-tolerant scheduling on multiprocessors, a relatively restricted fault model is considered, assuming, for example, that

- the inter-arrival time of two faults must be separated by a minimum distance [GMM94, TKK95, LLMM99]
- at most one fault may affect each task [LLMM99, GMM94]
- the recovery operation is simply the re-execution (i.e., does not consider a different implementation of the same task) [CYKT07, KLR10]

For the proposed algorithm, tolerating a maximum of f task errors within each possible interval of length D_{max}, where D_{max} is the largest relative deadline of a constrained-deadline sporadic task set is considered. In addition, tolerating at most ρ permanent processor failures during the life time of the system is also considered in the fault model. The assumed fault model does not put any restriction between the occurrences of consecutive task errors or processor failures. Any job of any task may suffer from multiple errors at any time. The backups of each task could simply be the re-execution of the primary or execution of a diverse implementation of the task.

The rest of this chapter is organized as follows: Section 8.2 presents related work. Section 8.3 presents the system models and the FTGS algorithm. In Section 8.4, the fault-tolerant schedulability problem is formally stated. The fault-tolerant global schedulability analysis considering only task errors is presented in Section 8.5–8.7. This analysis is then extended for tolerating processor failure in Section 8.8. Finally, Section 8.10 concludes the chapter.

8.2 Related Work

The fault-tolerant partitioned scheduling algorithms are traditionally based on Primary-Backup (PB) paradigm with the main aim for tolerating permanent processor failures [GMM94, TKK95, BMR99, CYKT07, KLR10]. In PB approach, each task is considered to have a primary and one or more backups. The primary and backups of each task are statically assigned (partitioned) to different processors at design time. Both task errors and processor failures are tolerated in the same way — by executing the backup of the affected task on a *different* processor.

A backup may be *active* or *passive*. An active backup *always* executes regardless of any error in its corresponding primary while a passive backup *only* executes after

the primary fails. Active backups are always executed even if the primary encounters no fault. Active backup policy utilizes more processing resource and energy but can provide better fault-tolerance for low-laxity (shorter deadline) tasks. In contrast, passive backup policy consumes less processing resource but may not provide enough fault-tolerance for the low-laxity tasks. Considering the wide ranges of resource-constrained embedded real-time systems, passive backups is considered for FTGS algorithm: the backup is only executed if an error is detected.

The work in [BT83] considers the allocation of a set of periodic tasks to a number of processors by assuming the same WCET of the r backups and does not consider minimizing the number of processors. The works in [OS94, OS95a] consider allocation of primary and multiple backups using RM first-fit [OS94] and RM next-fit [OS95a] heuristics. Both these algorithms requires at least twice the number of processors than that of required for some optimal allocation algorithm. The work in [CYKT07] proposes efficient allocation algorithm by simple modification of the first-fit, best-bit and worst-fit heuristics for minimizing the number of processors require to successfully assign a task set where each task has fixed number of replicas.

The works in [OS94, BMR99, CYKT07] consider active backups to tolerate only processor failures based on partitioned scheduling of *strictly* periodic real-time tasks. The task allocation algorithm proposed by Oh and Son in [OS94] considers multiple diverse backups of each task while the algorithm proposed by Chen *et al.* in [CYKT07] considers the backups simply as replicated copies of the primary. The task allocation algorithm proposed by Bertossi *et al.* in [BMR99] is based on RM first-fit bin packing heuristic to assign the primary and exactly *one* backup of each periodic task to the processors. Multiple active backups of the periodic tasks are considered by Kim *et al.* in [KLR10] for tolerating multiple processor failures; however, the backups are duplicates of the primary. None of these works consider sporadic task model and do not explicitly address the issue of tolerating only task errors. Task assignment with replication to achieve fault tolerance in heterogeneous processor processors are considered [ZQQ11, EB08].

Fault-tolerant scheduling of aperiodic tasks based on PB approach is proposed in [GMM94, TKK95]. Instead of considering active backup, passive backup [GMM94] or partially-active backup [TKK95] are found to be effective for fault tolerance. Moreover, in order to efficiently utilize the processors, the scheduling algorithms in [GMM94, TKK95] consider backup-backup *overloading* and backup *deallocation* techniques. In backup-backup overloading, two backup copies of two different primary copies overlapped in time on the same processor if their corresponding primaries are assigned in two different processors. Primary-backup overloading is considered in [AOSM01] and shown to have better schedulability than backup-backup overloading. In primary-backup overloading, the primary of a task can be scheduled onto the same or overlapping time interval with the backup of another task on a processor. These works considers only one backup copy for each task and assumes that there is a minimum separation interval between occurrences of consecutive processor failures. The work in [BFM97] consider RM first-fit policy for allocating periodic tasks to tolerate one processor failure using

PB approach by determining whether a task should use active or passive backup. The idea of [BFM97] is augmented with backup deallocation and overloading for implicit deadline task set in [BMR99]. To tolerate more processors failure at a certain time, the processors are statically [MM98] and dynamically [AOMS00] divided into disjoint logical groups such that one processor failure can be tolerated in each group.

There are very few works that have addressed fault-tolerance for global scheduling [BGJ06, LLMM99]. Fault-tolerant global scheduling based on probabilistic fault model is proposed by Berten *et al.* for global EDF scheduling in [BGJ06]. The algorithm in [BGJ06] considers simple re-execution of the tasks to tolerate only task errors based on EDF^k scheduling that is proposed by Goossens *et al.* in [GFB03]. The task model used in [BGJ06] is periodic and the deadline of each task is considered to be equal to its period. The pFair scheduling proposed by Baruah *et al.* in [BCPV96] for periodic task model is augmented with fault tolerance by Liberato *et al.* in [LLMM99]. However, the work in [LLMM99] considers exactly one backup for each task. Moreover, the schedulability test in [LLMM99] requires that there is a minimum separation between the occurrences of two consecutive task errors. There is no work that addresses fault-tolerant global scheduling that considers sporadic task model, fixed-priority, deadline of the tasks being less than or equal to the periods, and considers both processor failures and task errors using multiple and diverse backups. The proposed FTGS scheduling algorithm presented in this chapter possesses all these characteristics.

8.3 System Models and the FTGS Scheduling

Fault-tolerant scheduling of a set of constrained-deadline sporadic tasks on a multiprocessor platform consisting of m identical processors/cores is considered. The task and fault models for FTDM scheduling are presented in Section 3.1 and Section 3.3, respectively. The salient features of the models are reiterated here for better readability. A set of n constrained-deadline sporadic tasks $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ is considered, where each task $\tau_i \in \Gamma$ is characterized by WCET C_i, relative deadline D_i, and period T_i. A number of f task errors due to a variety of hardware and software faults that may occur within each of the all possible time intervals of length D_{max} is considered. Within any time interval of length D_{max}, the f task errors may occur in the same jobs or may occur in different jobs of different tasks. Each task has one primary and f backups. The WCET of the primary of task τ_i is C_i and the WCET of each of the f backups of task τ_i is denoted by E_i^k where $k = 1, 2, \ldots f$.

When there are $(f - c)$ task errors affecting the primary and subsequent $(f - c - 1)$ backups of the same job of task τ_i, the total execution requirement of the job of this task is denoted by $C_i^{(f\,-\,c)}$ and recursively calculated using Eq. (8.1) as follows:

$$C_i^{(f\,-\,c)} = \begin{cases} C_i & \text{if } (f-c) = 0 \\ E_i^{(f-c)} + C_i^{(f\,-\,c\,-\,1)} & \text{if } (f-c) > 0 \end{cases} \tag{8.1}$$

where $E_i^{(f-c)}$ is the WCET of the $(f - c)^{\text{th}}$ backup of task τ_i. Thus, starting from $k =$

$f, (f-1), \ldots 0$, all the values $C_i^0, C_i^1, \ldots C_i^f$ for task τ_i can be recursively calculated using total $O(f)$ addition operations (and for all tasks $O(n \cdot f)$ addition operations are needed). Note that the WCET, relative deadline, and inter-arrival time of each task τ_i must satisfy $C_i^f \leq D_i \leq T_i$.

The FTGS algorithm does not only capable of tolerating task errors but also can mitigate the effect of permanent processor failures. The FTGS algorithm considers mitigating the effect of ρ permanent processor failures during the lifetime of the system. The effect of processor failure is mitigated in FTGS algorithm by executing the backup of the affected task on a non-faulty processor. The backup in such case may be the re-execution of the primary.

Fault-Tolerant Mechanism and Algorithm FTGS: Each sporadic task generates an infinite number of jobs having a minimum inter-arrival time between successive jobs. The fault-tolerant mechanism based on time-redundancy for FTGS scheduling works as follows. For each job of a task, the primary executes first. Whenever a task error or processor failure is detected, the first backup of the affected task becomes ready to execute. The priority of the backup is same as that of its primary. Again, a task error or processor failure may be detected during the execution of the backup which in turn would trigger the execution of next backup and so on.

The scheduler is always made aware of all the non-faulty processors in the system. Such awareness can be achieved using *fail-signaled* processors. Once a processor failure is detected, the FTGS scheduler never dispatches any task to this faulty processor. Moreover, if a task was dispatched to this faulty processor, then the backup of the affected task becomes ready for execution. The FTGS scheduler stores all the ready (i.e., released but not completed) tasks in a global queue and dispatches the m highest priority tasks from this queue on m processors, possibly by preempting, if any, the execution of a lower priority task. The FTGS scheduling is based on global FP scheduling paradigm. Similar to the uniprocessor FTDM scheduling algorithm, it is assumed for FTGS algorithm that a task error is detected at the end of execution of the primary or backup. There is no fault propagation: one fault is assumed to affect at most one job either a primary or a backup. And, any primary or backup is assumed to be affected by at most one fault.

8.4 Problem Statement

In this chapter, the following problem is addressed:

> **Are all the deadlines of sporadic task set Γ met on m processors using FTGS scheduling if there are maximum f task errors within each of all possible time intervals of length D_{max} and a maximum of ρ processors failures during the lifetime of the system?**

Note that the maximum number of task errors within any time interval of length D_k is also f, for $k = 1, 2, 3, \ldots n$, because $D_k \leq D_{max}$. Following this, the problem statement for tolerating *only* task errors can be re-written as:

Are all the deadlines of task τ_k met on m processors using FTGS scheduling if there are maximum f task errors within any time interval of length D_k, for $k = 1, 2, \ldots n$?

In Sections 8.5–8.7, this later schedulability problem regarding tolerating only task errors is addressed first by proposing an iterative schedulability test — the schedulability of the entire task set is given in terms of a schedulability test for each of the lower priority task. Then this schedulability test is extended in Section 8.8 for mitigating the effect of ρ permanent processors failures.

8.5 Analysis for Tolerating Task Errors

The schedulability analysis presented in this and the following two sections derives a schedulability test for task $\tau_k \in \Gamma$ by assuming that all the higher priority tasks in HP_k meet their deadlines using FTGS scheduling. Then it follows that, if this test is satisfied for all the lower priority tasks in Γ (an iterative test), then the entire task set Γ is schedulable using FTGS algorithm. The proposed schedulability analysis in this chapter follows the same multiprocessors schedulability analysis framework proposed by Baker in [Bak06]: a *necessary condition* whenever any job of task τ_k misses its deadline is derived. Consequently, if this condition is not satisfied, then all the jobs of task τ_k meet their deadlines. When analyzing the schedulability of task τ_k, the occurrences of at most f task errors within any interval of length D_k is considered. In other words, the schedulability test for the FTGS scheduling algorithm is derived based on deadline-based analysis. In Section 8.8, this schedulability analysis for tolerating processor failures is extended.

Consider a generic job J_k of task τ_k. By generic it means that J_k represents an arbitrary job of task τ_k. The interval $[r_k, d_k]$ is called the *scheduling window* of job J_k. Note that the length of the scheduling window of any job of task τ_k is D_k. The ***computation load*** within the scheduling window of job J_k is defined to be equal to the cumulative length of the intervals during which job J_k is ready but not executing (interference) plus the total execution requirement of job J_k. Notice that the lower priority tasks $\tau_{k+1}, \ldots \tau_n$ do not contribute to the computation load since they can not interfere the execution of task τ_k in fixed-priority scheduling.

Consider the FTGS schedule of the tasks in set $(\mathsf{HP}_k \cup \{\tau_k\})$ such that all the jobs of tasks in HP_k meet their deadlines while the job J_k misses its deadline at d_k. The computation load within the scheduling window of job J_k must exceed D_k *if and only if* job J_k misses its deadline. Without loss of generality, job J_k is considered as a *critical job* in the sense that task τ_k is not feasible, if and only if, job J_k is not feasible using FTGS scheduling. It will be evident later from the schedulability analysis that it is not needed to know where in the schedule this critical job J_k is released. If the computation load of this critical job J_k within its scheduling window is not greater than D_k, then all the jobs of task τ_k meet deadlines, and conversely.

The computation load in the scheduling window of job J_k has two contributing factors: *interference* of the higher priority jobs and *self-execution requirement* of job J_k. The interference due to the tasks in HP_k and self-execution requirement of job J_k depend on the number of task errors within $[r_k, d_k]$. Let there be a errors that affect the higher priority jobs within $[r_k, d_k]$ and there are b errors of job J_k within $[r_k, d_k]$ when job J_k misses its deadline. Because there are at most f errors any interval of length D_k, we must have $(a+b) \leq f$. The self execution requirement of job J_k is at most C_k^b since job J_k suffers from b errors (according to Eq. (8.1)).

The ***interference*** within $[r_k, d_k]$ due to all the higher priority jobs in HP_k is defined as the cumulative length of intervals during which tasks of set HP_k are executing and job J_k is ready but not executing. The interference on τ_k within the interval $[r_k, d_k]$, where the higher priority jobs in HP_k suffer from a errors in $[r_k, d_k]$, is denoted by $\overline{I}_k^a([r_k, d_k])$. Thus, if job J_k misses its deadline, then

$$\overline{I}_k^a([r_k, d_k]) + C_k^b > D_k \tag{8.2}$$

where $(a+b) \leq f$ and D_k is the length of the scheduling window $[r_k, d_k]$. Since $(a+b) \leq f$, the following inequality in Eq. (8.3) holds:

$$\max_{c=0}^{f} \left\{ \overline{I}_k^c([r_k, d_k]) + C_k^{(f-c)} \right\} \geq \overline{I}_k^a([r_k, d_k]) + C_k^b \tag{8.3}$$

Therefore, from Eq. (8.2) and Eq. (8.3), it follows that if job J_k misses its deadline, then

$$\max_{c=0}^{f} \left\{ \overline{I}_k^c([r_k, d_k]) + C_k^{(f-c)} \right\} > D_k \tag{8.4}$$

The inequality in Eq. (8.4) is a **necessary** unschedulability condition for task τ_k. However, computing the interference $\overline{I}_k^c([r_k, d_k])$ of the higher priority jobs on job J_k within $[r_k, d_k]$ is difficult. This is because it is not known where in the schedule the job J_k is released. In other words, the *critical instant* — the job of task τ_k that suffers the maximum interference — is *unknown* for global multiprocessor scheduling (please see Example 3.1 and the discussion in page 36). The problem of not knowing the critical instant for determining the interference on a lower priority job is sidetracked by finding a safe upper bound on the interference due to the tasks in HP_k for global (non fault-tolerant) multiprocessor scheduling [Bak06, BCL09, GSYY09, DB09]. In order to find such an upper bound on actual interference, the upper bound on the *total interfering workload* which is the sum of the upper bounds of *interfering workload* of each of the tasks in HP_k has to be determined.

The ***interfering workload*** within $[r_k, d_k]$ of a higher priority task τ_i in HP_k is defined as the cumulative length of intervals during which task τ_i is executing and job J_k is ready but not executing. The ***total interfering workload*** within $[r_k, d_k]$ of all the higher priority tasks in a set HP_k is defined as the sum of the interfering workload of each task in set HP_k within $[r_k, d_k]$. Notice that the total interfering workload within $[r_k, d_k]$ of

the tasks in HP_k is equal to $(m \cdot \overline{I}_k^c([r_k, d_k]))$. This is because when task τ_k is interfered, all the m processors are simultaneously busy executing the higher priority tasks. The idea from [Bak06, BCL09, GSYY09, DB09] in finding the safe upper bound on the total interfering workload is adopted for the proposed fault-tolerant schedulability analysis of FTGS in this chapter. Deriving such an upper bound on total interfering workload does not require us to know the released time of job J_k in the fault-tolerant schedule. However, the pessimism in deriving the safe upper bound on total interfering workload needs to be reduced as much as possible in order to derive an effective sufficient schedulability test based on necessary unschedulability test.

The upper bound on the total interfering workload in $[r_k, d_k]$ due to all the higher priority tasks in HP_k, where c errors affect the higher priority jobs in $[r_k, d_k]$, is denoted by $\mathtt{I}_k^c(D_k)$. Thus, the following inequality holds:

$$\mathtt{I}_k^c(D_k) \geq m \cdot \overline{I}_k^c([r_k, d_k]) \tag{8.5}$$

Since interference $\overline{I}_k^c([r_k, d_k])$ is an integer, it follows that

$$\left\lfloor \frac{\mathtt{I}_k^c(D_k)}{m} \right\rfloor \geq \overline{I}_k^c([r_k, d_k]) \tag{8.6}$$

Thus from Eq. (8.4) and Eq. (8.6), if job J_k misses its deadline, then the following holds:

$$\max_{c=0}^{f} \left\{ \left\lfloor \frac{\mathtt{I}_k^c(D_k)}{m} \right\rfloor + C_k^{(f-c)} \right\} > D_k \tag{8.7}$$

The schedulability test proposed in this chapter for FTGS scheduling is based on the necessary unschedulability condition in Eq. (8.7) and needs to find the value of $\mathtt{I}_k^c(D_k)$ for all $c = 0, 1, \ldots f$. In Section 8.6, the upper bound on interfering workload of each higher priority task τ_i in HP_k is computed. The upper bound on interfering workload of all the tasks in HP_k are combined to find the value of $\mathtt{I}_k^c(D_k)$ in Section 8.7. Based on the value of $\mathtt{I}_k^c(D_k)$, the sufficient schedulability tests for FTGS scheduling is proposed.

8.6 Calculating Interfering Workload

The interfering workload of each task τ_i in HP_k is determined in two steps. First, an upper bound on the ***workload*** of each task τ_i in set HP_k within $[r_k, d_k]$ is determined (Subsection 8.6.1). Second, an upper bound on the interfering workload of each task τ_i within $[r_k, d_k]$ is calculated based on the upper bound on τ_i's workload within $[r_k, d_k]$ (Subsection 8.6.2). The value of the upper bound on the total interfering workload (i.e., $\mathtt{I}_k^c(D_k)$) is calculated in Section 8.7 by combining the upper bounds on interfering workload of all the tasks τ_i in HP_k in order to derive the schedulability test of the lower priority task τ_k.

8.6.1 Workload of task τ_i

The *workload* of task τ_i within an interval $[x, y]$ is the amount of time task τ_i executes in $[x, y]$. The work done by task τ_i in $[x, y]$ can be divided into three parts:

1. **Carry-in**: the contribution of at most one job (*called, carry-in job*) with release time earlier than x and deadline in $[x, y]$.
2. **Body**: the contribution of the jobs (*called, body jobs*) with both release time and deadline in $[x, y]$.
3. **Carry-out**: the contribution of at most one job (*called, carry-out job*) with release time in $[x, y]$ and deadline after y.

Finding the actual workload of a sporadic task τ_i in $[x, y]$ requires to consider all possible release times for all of its jobs in $[x, y]$. Instead, an upper bound on the workload of task τ_i within $[x, y]$ is calculated. The upper bound on the workload is computed based on the workload of each of the parts: carry-in, body, and carry-out job of task τ_i in $[x, y]$.

Task τ_i is called a *carry-in* task (CI-task) if task τ_i is considered to have carry-in work within the interval $[x, y]$; otherwise, task τ_i is called a *non carry-in* task (NC-task). The length of the interval $[x, y]$ id denoted as L where $L = (y - x)$. It is determined later whether task τ_i must be a CI-task or NC-task. The following notations are used to denote the carry-in and non carry-in workload of task τ_i within any interval of length L:

- $\mathtt{WNC}_i^g(\mathtt{L}, \xi)$ denotes the upper bound on the *non carry-in workload* of task τ_i in any interval of length L such that there are g errors of task τ_i in $[x, y]$ and the set ξ contains all the body and carry-out jobs of NC-task τ_i in $[x, y]$.
- $\mathtt{WCI}_i^g(\mathtt{L}, \xi)$ denotes the upper bound on the *carry-in workload* of task τ_i in any interval of length L such that there are g errors of task τ_i in $[x, y]$ and the set ξ contains all the carry-in, body and carry-out jobs of CI-task τ_i in $[x, y]$.

The calculation of $\mathtt{WNC}_i^g(\mathtt{L}, \xi)$ and $\mathtt{WCI}_i^g(\mathtt{L}, \xi)$ are presented next.

Calculating $\mathtt{WNC}_i^g(\mathtt{L}, \xi)$

Since task τ_i is a NC-task, there is no carry-in work of task τ_i in $[x, y]$. In order to find a safe upper bound on the workload of NC-task τ_i in $[x, y]$, it is needed to consider the densest possible packing of jobs of task τ_i in $[x, y]$. In such case, the released time of the first job of task τ_i in $[x, y]$ coincides with x. Without loss of generality, it is considered that job J_i^{p+1} of task τ_i has its release time exactly at the beginning of the interval $[x, y]$ and the subsequent jobs of τ_i are released as early as possible (see Figure 8.1).

Considering the densest possible packing of jobs of NC task τ_i, there are at most $\lfloor \frac{\mathtt{L}}{T_i} \rfloor$ body jobs and one carry-out job released within the interval $[x, y]$ of length L. Note that all the body jobs have their deadlines within the interval $[x, y]$ while the deadline of the carry-out job is outside the interval. Therefore, the maximum amount of work completed by the carry-out job in $[x, y]$ is upper bounded by $(\mathtt{L} - \lfloor \frac{\mathtt{L}}{T_i} \rfloor T_i)$.

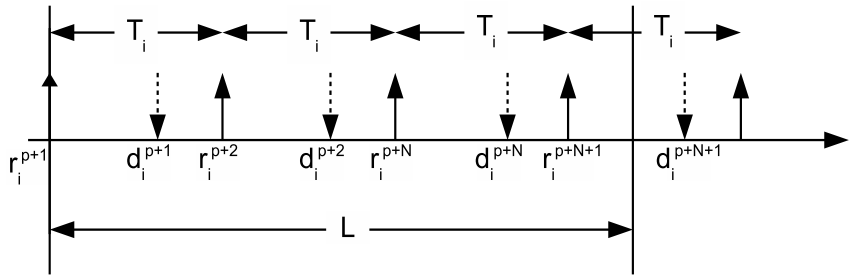

Figure 8.1: *Densest possible packing of the jobs of NC task τ_i within an interval of length L. The up-arrow and down-arrow are the released time and deadline of the jobs of task τ_i, respectively.*

The set of body jobs of the NC-task τ_i within $[x, y]$ are $\{J_i^{p+1}, \ldots J_i^{p+N}\}$ where $N = \lfloor \frac{\mathsf{L}}{T_i} \rfloor$ if $L \geq T_i$, otherwise there is no body job. The carry-out job is J_i^{p+N+1} if T_i is not an integer multiple of L, otherwise, there is no carry-out job. Therefore, $\xi = \{J_i^{p+1}, \ldots J_i^{p+N}, J_i^{p+N+1}\}$.

In order to find the value of $\mathsf{WNC}_i^g(\mathsf{L}, \xi)$, the worst-case occurrences of g errors affecting the primary and backups of the jobs in set ξ has to be determined. Aydin's work in [Ayd07] considered dynamic programming to compute the workload of a collection of aperiodic tasks scheduled using EDF on uniprocessor such that the aperiodic tasks suffer from a particular number of errors. Inspired by the work in [Ayd07], the value of $\mathsf{WNC}_i^g(\mathsf{L}, \xi)$ is computed based on the workload of each job in ξ. Since the jobs in set $\{J_i^{p+1}, \ldots J_i^{p+N}\}$ are from the same task τ_i, it follows that

$$\mathsf{WNC}_i^g(\mathsf{L}, \{J_i^{p+1}\}) = \ldots = \mathsf{WNC}_i^g(\mathsf{L}, \{J_i^{p+N}\}) = C_i^g \tag{8.8}$$

where C_i^g is given according to Eq. (8.1). The Eq. (8.8) essentially calculates workload of each individual job such that there are g errors effecting this particular job.

It is pointed out earlier that the maximum amount of carry-out work within the interval of length L is limited to $(\mathsf{L} - \lfloor \frac{\mathsf{L}}{T_i} \rfloor T_i)$. Thus, $\mathsf{WNC}_i^g(\mathsf{L}, \{J_i^{p+N+1}\})$ is given as follows:

$$\mathsf{WNC}_i^g(\mathsf{L}, \{J_i^{p+N+1}\}) = min\left\{ C_i^g \ , \left(\mathsf{L} - \left\lfloor \frac{\mathsf{L}}{T_i} \right\rfloor T_i \right) \right\} \tag{8.9}$$

In order to evaluate $\mathsf{WNC}_i^g(\mathsf{L}, \xi)$, the worst-case occurrences of g errors within $[x, y]$, affecting the jobs in $\{J_i^{p+1}, \ldots J_i^{p+N}, J_i^{p+N+1}\}$, has to be considered. The value of $\mathsf{WNC}_i^g(\mathsf{L}, \xi)$ is maximum, for some q, such that there are q errors of job $\{J_i^{p+1}\}$ and there ere $(g - q)$ errors of the jobs in set $\{J_i^{p+2}, \ldots J_i^{p+N+1}\}$ where $0 \leq q \leq g$. Thus,

the value of $\mathtt{WNC}_i^g(\mathtt{L}, \xi)$ is recursively given as follows:

$$\mathtt{WNC}_i^g(\mathtt{L}, \xi) = \mathtt{WNC}_i^g(\mathtt{L}, \{J_i^{p+1}\} \cup \{J_i^{p+2}, \ldots J_i^{p+N+1}\}) = \max_{q=0}^{g} \left\{ \mathtt{WNC}_i^q(\mathtt{L}, \{J_i^{p+1}\}) + \mathtt{WNC}_i^{(g-q)}(\mathtt{L}, \{J_i^{p+2}, \ldots J_i^{p+N+1}\}) \right\} \quad (8.10)$$

Calculating $\mathtt{WCI}_i^g(\mathtt{L}, \xi)$

Since task τ_i is CI-task, there is carry-in work of task τ_i in $[x, y]$. In such case, the released time of the first job of task τ_i in $[x, y]$ is earlier than x and its deadline is after x. Lets say the job J_i^p of CI-task τ_i is the carry-in job in $[x, y]$. Also let A is the set of body and carry-out jobs in $[x, y]$. Therefore, $\xi = \{J_i^p\} \cup A$. In order to find the upper bound on the workload of CI-task τ_i within $[x, y]$, the densest possible packing of the carry-in, body and carry-out jobs has to be considered. (Remember that L is the length of the interval $[x, y]$).

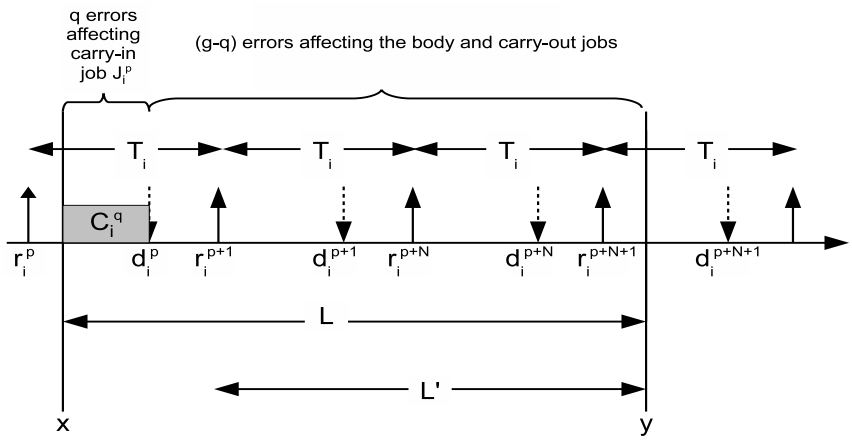

Figure 8.2: *The carry-in job J_i^p suffers from q errors and executes for C_i^q time units starting from the beginning of L. The subsequent body and carry-out jobs are released as early as possible within an interval of length $\mathtt{L}' = \mathtt{L} - (C_i^q + (T_i - D_i))$ and are subjected to $(a - q)$ errors.*

The value of $\mathtt{WCI}_i^g(\mathtt{L}, \xi)$ is maximum, for some q, such that there are q errors of the carry-in job J_i^p and the remaining $(g - q)$ errors affect the body and carry-out jobs in A where $0 \leq q \leq g$. The job J_i^p executes for C_i^q time units if there are q errors of this job. The workload $\mathtt{WCI}_i^g(\mathtt{L}, \xi)$ within $[x, y]$ is maximized for some q (depicted in Figure 8.2), if the following two conditions are satisfied:

- **C1:** the carry-in job starts execution exactly at time x and finishes its execution exactly at its deadline which is $d_i^p = x + C_i^q$ (see the shaded execution of the carry-in job in Figure 8.2), and

- **C2:** the subsequent jobs (i.e., body and carry-out jobs) are released and execute as early as possible such that $(g - q)$ errors affect the body and carry out jobs,

where $0 \leq q \leq g$.

To show that these two conditions (i.e., **C1** and **C2**) result in maximum workload in $[x, y]$, it will be shown that for any leftward shift of the interval $[x, y]$ up to T_i time units, the amount of workload within the interval $[x, y]$ does not increase as long as τ_i has carry-in contribution. Note that since the situation is periodic (i.e., jobs arrives as compactly as possible), shifting the interval for exactly T_i time units again produces the same situation as in Figure 8.2. Therefore, any leftward shift of the interval for at most T_i time units is considered.

Consider leftward shift of interval $[x, y]$ up to $(x - r_i^p)$ time units. In such case the carry-in contribution can not increase and the carry-out can only decrease. Now consider a leftward shift of $[x, y]$ for more than $(x - r_i^p)$ time units but less than T_i time units. Any leftward shift of $[x, y]$ by Δ time units is equivalent to shifting $[x, y]$ rightward for $(T_i - \Delta)$ time units. Thus, the leftward shift of $[x, y]$ for more than $(x - r_i^p)$ time units but less than T_i time units is equivalent to shifting $[x, y]$ rightward for more than 0 time units but less than $T_i - (x - r_i^p)$ time units. Any rightward shift of the interval $[x, y]$ cause the carry-in work to decrease while the carry-out work can only be increased by the same amount as long as there is carry-in contribution in $[x, y]$. Evidently, if τ_i is a `CI`-task, then the workload within the interval $[x, y]$ is maximum if the conditions **C1** and **C2** are satisfied.

The workload of the carry-in job is given as C_i^q according to Eq. (8.1). It is evident from Figure 8.2 that the body and carry-out jobs are released within the interval $[r_i^{p+1}, y]$. This situation is same as in Figure 8.1 where task τ_i is a `NC` task within $[r_i^{p+1}, y]$. The length of the interval $[r_i^{p+1}, y]$, denoted by L', is given as $\mathtt{L}' = \mathtt{L} - (C_i^q + (T_i - D_i))$. And, according to Eq. (8.10), the value of $\mathtt{WNC}_i^{(g-q)}(\mathtt{L}', A)$ is the worst-case workload of the body and carry-out jobs within the interval $[r_i^{p+1}, y]$ such that these body and carry-out jobs are subjected to $(g - q)$ errors. Thus, the value of $\mathtt{WCI}_i^g(\mathtt{L}, \xi)$ is given as follows:

$$\mathtt{WCI}_i^g(\mathtt{L}, \xi) = \max_{q=0}^{g} \left\{ C_i^q + \mathtt{WNC}_i^{(g-q)}(\mathtt{L}', A) \right\} \tag{8.11}$$

where $\xi = (\{J_i^p\} \cup A)$, set A is the collection of body and carry-out jobs in $[r_i^{p+1}, y]$, $\mathtt{L}' = \mathtt{L} - (C_i^q + (T_i - D_i))$ and C_i^q is given using Eq. (8.1).

Note that the value of $\mathtt{WCI}_i^g(\mathtt{L}, \xi)$ is greater than or equal to $\mathtt{WNC}_i^g(\mathtt{L}, \xi)$. This is because shifting the interval $[x, y]$ leftward for exactly $(x - r_i^p)$ time units in Figure 8.2 produces the same scenario as in Figure 8.1 for `NC`-task, and such leftward shift can only reduce the workload within the interval $[x, y]$. In next subsection, the upper bound on interfering workload of task τ_i on job J_k based on workload of task τ_i in $[r_k, d_k]$ is determined. The value of carry-in workload may be reduced further by exploiting the slack of the carry-in higher priority task similar to the approach in [BCL09]; however, this issue is not addressed in this thesis.

8.6.2 Interfering Workload of task τ_i

The upper bound on interfering workload of each higher priority task τ_i based on the upper bound on the workload of task τ_i is calculated. Similar to workload, the carry-in and non carry-in interfering workload of task τ_i subjected to g errors in $[r_k, d_k]$ are defined as follows:

- $\mathtt{INC}_i^g(D_k, c)$ denotes the upper bound on the interfering workload of NC-task τ_i in any interval $[r_k, d_k]$ of length D_k such that there are g errors of task τ_i and there are c errors of all the higher priority tasks (including task τ_i) of τ_k in $[r_k, d_k]$.

- $\mathtt{ICI}_i^g(D_k, c)$ denotes the upper bound on the interfering workload of CI-task τ_i in any interval $[r_k, d_k]$ of length D_k such that there are g errors of task τ_i and there are c errors of all the higher priority tasks (including task τ_i) of τ_k in $[r_k, d_k]$.

In both $\mathtt{INC}_i^g(D_k, c)$ and $\mathtt{ICI}_i^g(D_k, c)$, it is assumed that there are c errors affecting all the tasks in $\mathtt{HP}_k$ within $[r_k, d_k]$ where g errors, $g \leq c$, exclusively affect the higher priority task $\tau_i \in \mathtt{HP}_k$.

A straightforward upper bound on the interference of each task τ_i in $[r_k, d_k]$ is the upper bound on the workload of each task τ_i in $[r_k, d_k]$. However, this way of bounding the interference using the upper bound on the workload may be pessimistic as pointed out in [Bar07, BC07, BCL09] for non-fault-tolerant global multiprocessor scheduling. This fact is also true for the fault-tolerant schedulability analysis of FTGS scheduling as is shown below.

If job J_k misses its deadline when the higher priority jobs suffer from c errors and job J_k suffer from $(f - c)$ errors, the amount of work completed by job J_k within $[r_k, d_k]$ is strictly less than $C_k^{(f-c)}$. If job J_k misses its deadline, then all the m processors simultaneously execute jobs of the higher priority tasks for strictly more than $(D_k - C_k^{(f-c)})$ time units. Therefore, if job J_k suffers enough interference in $[r_k, d_k]$ to miss its deadline, then it is sufficient to consider the interfering workload of each task τ_i limited to at most $(D_k - C_k^{(f-c)} + 1)$. Thus, the value of $\mathtt{INC}_i^g(D_k, c)$ and $\mathtt{ICI}_i^g(D_k, c)$ are given as follows:

$$\mathtt{INC}_i^g(D_k, c) = min\{\mathtt{WNC}_i^g(D_k, \xi), D_k - C_i^{(f-c)} + 1\} \tag{8.12}$$

$$\mathtt{ICI}_i^g(D_k, c) = min\{\mathtt{WCI}_i^g(D_k, \xi), D_k - C_i^{(f-c)} + 1\} \tag{8.13}$$

Similar to workloads, it is not difficult to see that the carry-in interference $\mathtt{ICI}_i^g(D_k, c)$ is greater than or equal to the non carry-in interference $\mathtt{INC}_i^g(D_k, c)$ for task τ_i. Given the values of $\mathtt{INC}_i^g(D_k, c)$ and $\mathtt{ICI}_i^g(D_k, c)$ for all $\tau_i \in \mathtt{HP}_k$ and for all $g = 0, 1, \ldots c$, the value of combined interference $\mathtt{I}_k^c(D_k)$ is calculated in Section 8.7.

It will be discussed shortly that only a subset of all the higher priority tasks in $\mathtt{HP}_k$ are considered as CI tasks. However, such a subset must be selected such that the difference between its total carry-in interfering workload and total non carry-in interfering workload within $[r_k, d_k]$ is the largest in comparison to that of any other subset of the

higher priority tasks having the same cardinality. The following function and set definitions will be used to determine the set of carry-in tasks in next subsection.

Useful Definitions: Consider a subset Y of the task set HP_k such that within the scheduling window of J_k, there are g errors of the tasks in Y and there are c errors of the task in HP_k. Note that the occurrences of the g errors are part of the occurrences of the c errors since $Y \subseteq \mathsf{HP}_k$. We denote $\mathtt{DIFF}_g(Y, D_k, c)$ as the maximum difference between the

- total carry-in interfering workload by considering all tasks in Y as carry-in tasks, and
- total non carry-in interfering workload by considering all tasks in Y as no carry-in tasks

within the scheduling window of job J_k where there are g errors of the tasks in Y and there are c errors of the task in HP_k. If there is exactly one task in set Y, say $Y = \{\tau_i\}$, then $\mathtt{DIFF}_g(Y, D_k, c)$ is given as

$$\mathtt{DIFF}_g(\{\tau_i\}, D_k, c) = \mathtt{ICI}_g^i(D_k, c) - \mathtt{INC}_g^i(D_k, c) \tag{8.14}$$

If set Y has more than one task, say $Y = X \cup \{\tau_i\}$ where X is the set of all tasks in set Y except task τ_i, then the value of $\mathtt{DIFF}_g(Y, D_k, c)$ is maximized, for some q, such that there are q errors of the tasks in set X and there are $(g - q)$ errors of the task τ_i within the interval $[r_k, d_k]$, where $0 \leq q \leq g$. Thus, the value of $\mathtt{DIFF}_g(Y, D_k, c)$ can be recursively calculated as follows:

$$\begin{aligned}&\mathtt{DIFF}_g(Y, D_k, c) = \mathtt{DIFF}_g(X \cup \{\tau_i\}, D_k, c)\\ &= \max_{q=0}^{g} \left\{ \mathtt{DIFF}_q(X, D_k, c) + \mathtt{DIFF}_{(g-q)}(\{\tau_i\}, D_k, c) \right\}\end{aligned} \tag{8.15}$$

We define $\mathcal{Q}(S, a, \hat{m}, c)$ as a subset of the task set S such that $\mathcal{Q}(S, a, \hat{m}, c)$ has $\hat{m}$ tasks from set S and satisfies **Constraint C1** that is given for set $\mathcal{Q}(S, a, \hat{m}, c)$ as follows:

Constraint C1: The tasks in set $\mathcal{Q}(S, a, \hat{m}, c)$

- has m' tasks from set S,
- are subjected to the worst-case occurrences of a errors within $[r_k, d_k]$,
- where there are at most c errors that affect the tasks in set S within $[r_k, d_k]$, and
- the difference between
  - the total interfering workload of these $\hat{m}$ tasks considering each task in $\mathcal{Q}(S, a, \hat{m}, c)$ as a $\mathtt{CI}$-task, and
  - the total interfering workload of these $\hat{m}$ tasks considering each task in $\mathcal{Q}(S, a, \hat{m}, c)$ as a $\mathtt{NC}$-task

  is greater than or equal to that of any other subset of $\hat{m}$ tasks from set S.

Formally, if set $\mathcal{Q}(S, a, \hat{m}, c)$ satisfies **Constraint C1**, then for any other set B such that $B \subseteq S$ and $|B| = \hat{m}$, we have

$$\texttt{DIFF}_a(\mathcal{Q}(S, a, \hat{m}, c), D_k, c) \geq \texttt{DIFF}_a(\texttt{B}, D_k, c) \tag{8.16}$$

The definition of set $\mathcal{Q}(S, a, \hat{m}, c)$ is used in the next section to determine the set of carry-in tasks. Once the set of carry-in and non carry-in tasks are known, the interfering workloads of all tasks in $\texttt{HP}_k$ are combined to find $\texttt{I}_k^c(D_k)$.

8.7 Total Interfering Workload of the Tasks in $\texttt{HP}_k$

In order to find the upper bound on total interfering workload $\texttt{I}_k^c(D_k)$, the upper bound on interfering workload in $[r_k, d_k]$ of all the tasks in $\texttt{HP}_k$ have to be combined considering the worst-case occurrences of c task errors affecting the tasks in $\texttt{HP}_k$. Whether task τ_i should be considered as a CI or NC task has to be determined before combining the interfering workload of individual task. Based on Baruah's idea in [Bar07] for global EDF, it has already been shown in [GSYY09, DB11b] that for global fixed-priority scheduling, there are at most $(m-1)$ higher priority tasks that have carry-in work within the scheduling window of any lower priority job.

However, selecting the $(m-1)$ carry-in tasks from set $\texttt{HP}_k$ is challenging for two reasons: (i) there are $\binom{|\texttt{HP}_k|}{(m-1)} = \frac{(k-1)!}{(m-1)!\cdot(k-m)!}$ possible ways to select a subset of $(m-1)$ tasks from set $\texttt{HP}_k$, and more importantly, (ii) the carry-in or non carry-in interfering workload of each task τ_i depends on the number of errors affecting task τ_i which in turn depends on the worst-case occurrence of the c errors affecting all the tasks in $\texttt{HP}_k$ within $[r_k, d_k]$. To solve the problem of finding $(m-1)$ carry-in tasks efficiently, the algorithm, called FindCITasks, is proposed in Subsection 8.7.1 based on dynamic programming approach. Given the sets of carry-in and non carry-in tasks, the individual carry-in and non carry-in interfering workload of all tasks are combined to find $\texttt{I}_k^c(D_k)$ in Subsection 8.7.2. Finally, the schedulability test for FTGS algorithm based on this total interfering workload is proposed.

8.7.1 Finding Carry-in Set $\mathcal{Q}(S, a, \hat{m}, c)$

Recall that set $\mathcal{Q}(S, a, \hat{m}, c)$ is a subset of $\hat{m}$ tasks from set S and satisfies **Constraint 1**. In this subsection, an algorithm called FindCITasks that finds the set $\mathcal{Q}(S, a, \hat{m}, c)$ is proposed. Two cases are considered to find the set $\mathcal{Q}(S, a, \hat{m}, c)$: Case(i) $\hat{m} = 1$, and Case(ii) $\hat{m} > 1$.

Case(i) $\hat{m} = 1$**:** For this case, the aim is to find set $\mathcal{Q}(S, a, 1, c)$ such that **Constraint 1** is satisfied. The set $\mathcal{Q}(S, a, 1, c)$ is given as follows:

$$\mathcal{Q}(S, a, 1, c) = \{\tau_x\} \tag{8.17}$$

such that task τ_x satisfies Eq. (8.18)

$$\texttt{DIFF}_a(\{\tau_x\}, D_k, c) = \max_{\tau_i \in S} \left\{ \texttt{DIFF}_a(\{\tau_i\}, D_k, c) \right\} \tag{8.18}$$

which implies **Constraint 1** is satisfied for set $\mathcal{Q}(S, a, 1, c)$.

Case (ii) $\hat{m} > 1$**:** For this case, it is required to find from set S more than one carry-in tasks that are subjected to a errors within the scheduling window of job J_k. Two steps are considered to find such $\hat{m}$ number of tasks from set S:

- **Step 1** Find exactly one carry-in task from set S.
- **Step 2** Recursively find $(\hat{m}-1)$ carry-in tasks from set $(S - \{\tau_x\})$ where task τ_x is found in Step 1.

These two steps (i.e., **Step 1** and **Step 2**) have to consider the worst-case occurrences of a errors that can affect all these $\hat{m}$ tasks within the scheduling window of job J_k. The worst-case is determined by considering that $(a-\alpha)$ errors exclusively affect the task determined in Step 1, and the worst-case occurrences of α errors affecting the other $(\hat{m}-1)$ tasks determined in Step 2 for $\alpha = 0, 1, \ldots a$. For Step 1, the task affected by $(a-\alpha)$ errors is in the set $\mathcal{Q}(S, a-\alpha, 1, c)$ and can be determined using Eq. (8.17).

For Step 2, the other $(\hat{m}-1)$ tasks are selected from set $(S - \mathcal{Q}(S, a-\alpha, 1, c))$ considering an occurrences of α errors affecting these $(\hat{m}-1)$ tasks. This set of $(m'-1)$ tasks is given by $\mathcal{Q}(S', \alpha, \hat{m}-1, c)$ where $S' = (S - \mathcal{Q}(S, a-\alpha, 1, c))$. The tasks found in Step 1 and Step 2 for a particular α are given in set S_α as follows:

$$S_\alpha = \mathcal{Q}(S, a-\alpha, 1, c) \cup \mathcal{Q}(S', \alpha, \hat{m}-1, c) \tag{8.19}$$

where $S' = (S - \mathcal{Q}(S, a-\alpha, 1, c))$ and $0 \leq \alpha \leq a$. The set S_α is a potential candidate for set $\mathcal{Q}(S, a, \hat{m}, c)$ which must satisfy **Constraint C1** where $0 \leq \alpha \leq a$. Therefore, the set $\mathcal{Q}(S, a, \hat{m}, c)$ for $\hat{m} > 1$ is given as follows:

$$\mathcal{Q}(S, a, \hat{m}, c) = S_x \tag{8.20}$$

where set S_x satisfies

$$\texttt{DIFF}_a(S_x, D_k, c) = \max_{\alpha=0}^{a} \left\{ \texttt{DIFF}_a(S_\alpha, D_k, c) \right\} \tag{8.21}$$

which implies **Constraint 1** is satisfied for set $S_x = \mathcal{Q}(S, a, \hat{m}, c)$. The algorithm `FindCITasks`$(S, a, \hat{m}, c)$ in Figure 8.3 determines the set $\mathcal{Q}(S, a, \hat{m}, c)$ based on these two cases: case(i) $\hat{m} = 1$ (line 1–5), and case(ii) $\hat{m} > 1$ (line 6–16).

When $\hat{m} = 1$ (line 1–5), the task τ_x is selected from set S such that Eq. (8.18) is satisfied. The task τ_x is returned in line 4. When $\hat{m} > 1$ (line 6–16), the sets S_α for all $\alpha = 0, 1, \ldots a$ using Eq. (8.19) have to be determined first. Then, the set S_x which is equal to set $\mathcal{Q}(S, a, \hat{m}, c)$ is determined by evaluating Eq. (8.21). The for

Algorithm `FindCITasks`(S, a, $\hat{m}$, c)

1. If $(\hat{m} = 1)$ Then
2. Find task τ_x such that
3. $\texttt{DIFF}_a(\{\tau_x\}, D_k) = \max_{\tau_i \in S} \left\{ \texttt{DIFF}_a(\{\tau_i\}, D_k) \right\}$
4. Return $\{\tau_x\}$
5. End If
6. If $(\hat{m} > 1)$ Then
7. For $\alpha = 0$ to a
8. $\mathcal{Q}(S, a - \alpha, 1, c)$= `FindCITasks`$(S, a - \alpha, 1, c)$
9. $S' = S - \mathcal{Q}(S, a - \alpha, 1, c)$
10. $\mathcal{Q}(S', \alpha, \hat{m} - 1, c)$=`FindCITasks`$(S', \alpha, \hat{m} - 1, c)$
11. $S_\alpha = \mathcal{Q}(S, a - \alpha, 1, c) \cup \mathcal{Q}(S', \alpha, \hat{m} - 1, c)$
12. End For
13. Find task set S_x such that
14. $\texttt{DIFF}_a(S_x, D_k, c) = \max_{\alpha=0}^{a} \left\{ \texttt{DIFF}_a(S_\alpha, D_k, c) \right\}$
15. Return S_x
16. End If

Figure 8.3: *Pseudocode for finding carry-in tasks*

loop in line 7–12 runs a total of $(a + 1)$ times for the iterative variable $\alpha = 0, \ldots a$. For each value of α, the set $\mathcal{Q}(S, a - \alpha, 1, c)$ is determined by recursively calling `FindCITasks`$(S, a - \alpha, 1, c)$ in line 8. The set $S' = S - \mathcal{Q}(S, a - \alpha, 1, c)$ is determined in line 9. The rest of the $(\hat{m} - 1)$ tasks are determined in line 10 by recursively calling `FindCITasks`$(S', \alpha, \hat{m} - 1, c)$. Finally, the set S_α is determined in line 11.

The value of $\texttt{DIFF}_a(S_\alpha, D_k, c)$ can be determined using Eq. (8.15) for all $\alpha = 0, 1, \ldots a$. The set S_α that satisfies **Constraint 1** is the set $\mathcal{Q}(S, a, \hat{m}, c)$. The set $\mathcal{Q}(S, a, \hat{m}, c) = S_x$, that satisfies **Constraint 1** for some $0 \leq x \leq a$, is searched in line 13–14 and returned in line 15. The set of $(m - 1)$ carry-in tasks from set $\texttt{HP}_k$, where the carry-in tasks are affected by q errors and all tasks in $\texttt{HP}_k$ are affected by c errors, is $\mathcal{Q}(\texttt{HP}_k, q, m - 1, c)$ and can be determined by calling `FindCITasks`$(\texttt{HP}_k, q, m - 1, c)$.

The time complexity of algorithm `FindCITasks`$(S, a, \hat{m}, c)$ is now presented by assuming that the value of $\texttt{DIFF}_g(\{\tau_i\}, D_k, c)$ is known for all $\tau_i \in S$ and for all $g = 0, 1, \ldots a$. The base case, i.e., when $\hat{m} = 1$, in line 1–5 can be determined in $O(n)$ steps since there are at most n tasks in set S.

When $\hat{m} > 1$, the set S_α has to be determined for all $\alpha = 0, 1, \ldots a$. For each value of α, the base case in line 1–5 is evaluated total $\hat{m}$ times and there are total $(\hat{m} - 1)$ set difference operations in line 9. The time complexity for each call of the base case in line 1–5 is $O(n)$. The set difference in line 9 can be done in linear time since only one element is removed from set S. The set union in line 10 can be done in constant time

since one of the sets has only one element. Thus, for a particular α, the time complexity to determine the set S_α in line 11 is $O(n \cdot m)$.

The tasks in set S_α are potential candidates for the set of $\hat{m}$ carry-in tasks while the tasks in $(S - S_\alpha)$ are potential candidates for the set of non carry-in tasks, for $\alpha = 0, \ldots a$. The for loop in line 7–12 finds all the potential $\mathtt{CI}$ task sets $S_1, S_2 \ldots S_a$. Note that since $a \leq f$, the for loop in line 7–12 runs total $O(f)$ time. Thus, the time complexity to find all the sets $S_1, S_2 \ldots S_a$ in line 7–12 is $O(n \cdot m \cdot f)$.

It is not difficult to see that given the values of $\mathtt{DIFF}_g(\{\tau_i\}, D_k, c)$, for all $g = 0, 1, \ldots a$, evaluating $\mathtt{DIFF}_a(S_\alpha, D_k, c)$ using Eq. (8.15) has the time complexity of $O(n \cdot f)$ since there are at most n elements in S_α and the maximum operation in Eq. (8.15) needs at most g comparisons for set $X \cup \{\tau_i\}$ where $g \leq c \leq f$. Therefore, evaluating the value of $\mathtt{DIFF}_a(S_\alpha, D_k, c)$ in line 13-14 for all $\alpha = 0, \ldots a$ has time complexity $O(n \cdot f^2)$. Thus, the time complexity of the algorithm `FindCITasks` is $O(n \cdot m \cdot f + n \cdot f^2) = O(n \cdot f \cdot max\{m, f\})$ if the values of $\mathtt{DIFF}_g(\{\tau_i\}, D_k, c)$, for all $g = 0, 1, \ldots a$ and for all $\tau_i \in S$ are known.

8.7.2 Total Interfering Workload and Schedulability Test

The total interfering workload, i.e., the value of $\mathtt{I}_k^c(D_k)$ is computed by combining the upper bound on the interfering workload of all the higher priority tasks in HP_k. Recall that there are $(m-1)$ carry-in tasks in set HP_k. The worst-case occurrence of the c errors affecting the tasks in HP_k needs to consider the worst-case occurrence of q errors affecting the $(m-1)$ carry-in tasks and the worst-case occurrence of $(c-q)$ errors affecting the $(|\mathsf{HP}_k| - m + 1)$ non carry-in tasks for some q, $0 \leq q \leq c$. The set of $(m-1)$ carry-in task are given by $\mathcal{Q}(\mathsf{HP}_k, q, m-1, c)$ according to Eq. (8.20) and can be determined by calling `FindCITasks`$(\mathsf{HP}_k, q, m-1, c)$ for a particular q, $0 \leq q \leq c$. And the remaining $(|\mathsf{HP}_k| - m + 1)$ non carry-in tasks are given by set $(\mathsf{HP}_k - \mathcal{Q}(\mathsf{HP}_k, q, m-1, c))$.

Before combining the upper bound of the individual interfering workload of all the tasks in HP_k to find $\mathtt{I}_k^c(D_k)$, the following problem needs to be solved: ***Consider a set Y such that $Y \subseteq \mathsf{HP}_k$ and assume that it is already known whether τ_i is a $\mathtt{CI}$ task or $\mathtt{NC}$ task for each task $\tau_i \in Y$. What is the upper bound on the total interfering workload of the tasks in set Y on task τ_k within $[r_k, d_k]$ if the tasks in set Y suffer from g errors and all the tasks in HP_k suffers from c errors within $[r_k, d_k]$?***

The upper bound on the total interfering workload of the tasks in set Y on task τ_k within $[r_k, d_k]$ is denoted as $\mathtt{TI}^g(Y, D_k, c)$ such that there are g and c errors within $[r_k, d_k]$ affecting the tasks in sets Y and HP_k, respectively (note that the g errors are part of the c errors since $Y \subseteq \mathsf{HP}_k$). If there is exactly one task in set Y, such that $Y = \{\tau_i\}$, then $\mathtt{TI}^g(Y, D_k, c)$ is given as follows:

$$\mathtt{TI}^g(Y, D_k, c) = \begin{cases} \mathtt{ICI}_i^g(D_k, c) & \text{if } \tau_i \text{ is a } \mathtt{CI}\text{-task} \\ \mathtt{INC}_i^g(D_k, c) & \text{if } \tau_i \text{ is a } \mathtt{NC}\text{-task} \end{cases} \tag{8.22}$$

where $\mathtt{ICI}_i^g(D_k, c)$ and $\mathtt{INC}_i^g(D_k, c)$ are defined in Eq. (8.13) and Eq. (8.12), respec-

tively. Now consider the case where the set Y has more than one task, say $Y = X \cup \{\tau_i\}$, where X is the set of all the tasks in Y except task τ_i. The value of $\mathtt{TI}^g(Y, D_k, c)$ is maximized, for some q, if there are q errors of the tasks in set X and there are $(g-q)$ errors of the task τ_i, where $0 \leq q \leq g$. Thus, the value of $\mathtt{TI}^g(Y, D_k, c)$ can be recursively calculated as follows:

$$\begin{aligned}\mathtt{TI}^g(\mathrm{Y}, D_k, c) &= \mathtt{TI}^g(\mathrm{X} \cup \{\tau_i\}, D_k, c) \\ &= \max_{q=0}^{g} \left\{ \mathtt{TI}^q(\mathrm{X}, D_k, c) + \Psi \right\} \end{aligned} \tag{8.23}$$

$$\text{where} \quad \Psi = \begin{cases} \mathtt{ICI}_i^{g-q}(D_k, c) & \text{if } \tau_i \text{ is a } \mathtt{CI}\text{-task} \\ \mathtt{INC}_i^{g-q}(D_k, c) & \text{if } \tau_i \text{ is a } \mathtt{NC}\text{-task} \end{cases}$$

Recall that the upper bound on combined interference $\mathtt{I}_k^c(D_k)$ is sum of the upper bound of the individual interfering workload of the higher priority tasks in HP_k where these tasks are affected by c errors within $[r_k, d_k]$. The sum of the upper bounds of the individual interferences of the $\mathtt{CI}$ and $\mathtt{NC}$ tasks is maximum, for some q, where there are q errors of the $(m-1)$ carry-in tasks and there are the $(c-q)$ errors of the remaining $(|\mathsf{HP}_k| - m + 1)$ non carry-in tasks, $0 \leq q \leq c$. Thus, the value of $\mathtt{I}_k^c(D_k)$ is given as follows:

$$\mathtt{I}_k^c(D_k) = \max_{q=0}^{c} \left\{ \mathtt{TI}^q(A, D_k, c) + \mathtt{TI}^{g-q}(B, D_k, c) \right\} \tag{8.24}$$

where $A = \mathcal{Q}(\mathsf{HP}_k, q, m-1, c)$ is the set of $(m-1)$ carry-in tasks and $B = (\mathsf{HP}_k - A)$ is the set of non carry-in tasks. Note that the set A in Eq. (8.24) may be different for different values of q, $0 \leq q < c$. The value of $\mathtt{I}_k^c(D_k)$ given in Eq. (8.24) is the upper bound on the total interfering workload.

Sufficient Schedulability Test. Now based on the necessary unschedulability condition in Eq. (8.7), the following sufficient schedulability test in Theorem 8.1 for task τ_k follows:

Theorem 8.1. *A task $\tau_k \in \Gamma$ is schedulable using $\mathtt{FTGS}$ algorithm if*

$$\max_{c=0}^{f} \left\{ \left\lfloor \frac{\mathtt{I}_k^c(D_k)}{m} \right\rfloor + C_k^{(f-c)} \right\} \leq D_k \tag{8.25}$$

Proof. This Theorem is proved using contradiction. Assume that some job of task τ_k has missed its deadline while the condition in Eq. (8.25) holds. Remember that job J_k is a critical job in the sense that task τ_k is not feasible using $\mathtt{FTGS}$ if and only if J_k misses its deadline. Consequently, if at least one job of task τ_k misses its deadline, then job J_k also misses its deadline which implies that Eq. (8.7) holds (contradicts the fact that Eq. (8.25) holds!). □

The schedulability test of the *entire* task set Γ is given by iteratively applying Theorem 8.1 on each lower priority task τ_k for $k = (m+1), \ldots n$.

Corollary 8.1. *Sporadic task set* Γ *is feasible using* FTGS *algorithm if*

$$\max_{c=0}^{f} \left\{ \left\lfloor \frac{\boldsymbol{I}_k^c(D_k)}{m} \right\rfloor + C_k^{(f-c)} \right\} \leq D_k \tag{8.26}$$

for all $k = m+1, \ldots n$.

A concise notation for the iterative schedulability test of Corollary 8.1 is denoted by `FTGS-Test(`Γ, f, m`)`that when passed for a task set Γ guarantees that all the tasks in Γ meet their deadlines on m processors even if there are f task errors in any interval of length D_{max}. The Pseudocode to evaluate `FTGS-Test(`Γ, f, m`)`is given in Figure 8.4.

The algorithm in Figure 8.4 starts by calculating $C_k^{(f-c)}$ for all $k = 1, 2, \ldots n$ and for all $(f-c) = 0, 1, \ldots f$ in line 1–6 of Figure 8.4. In other words, the values of $C_k^0, C_k^1 \ldots C_k^f$ for all $k = 1, \ldots n$ is calculated in line 1–6. The for loop in line 7–33 runs total $(n-m)$ times and evaluates the schedulability condition in Eq. (8.26) for each of the lower priority tasks τ_k in each iteration for $k = m+1, \ldots n$. When evaluating the schedulability of task τ_k, the carry-in workload and non carry-in workload of each of the higher priority tasks are determined first in line 8–13. Then, the individual carry-in interfering workload, non carry-in interfering workload, and their difference for each higher priority task are determined in line 14–22. Finally, the total interfering workload of all the higher priority tasks are determined and Eq. (8.26) is evaluated in line 23–32.

The condition in line 28 checks whether the total computation load of the tasks in $\mathsf{HP}_k \cup \{\tau_k\}$ in any interval of length D_k exceeds D_k where task τ_k is affected by $(f-c)$ errors, the carry-in tasks are affected by q errors, and the non carry-in tasks are affected by $(c-q)$ error for all q, c and f such that $q \leq c \leq f$. If the answer is positive (computation load is greater than the length of the interval), then task τ_k can not be guaranteed to be schedulable and the algorithm returns "False" in line 29. If the condition at line 28 is never true, then the for loop at line 7–33 is exited, the algorithm returns "True" in line 34 and the entire task set is schedulable using the FTGS scheduling. The time complexity of `FTGS-Test(`Γ, f, m`)`is pseudo-polynomial as is shown next.

Time Complexity. Remember that the self execution time of all the n tasks when affected by $(f-c)$ errors can be determined in $O(n \cdot f)$ time. So, line 1–6 runs in $O(n \cdot f)$ time. The two nested for loops in line 8–13 determine the carry-in and non carry-in workload of each of the higher priority tasks when considering the schedulability of task τ_k. All the higher priority tasks are iterated using the iterative variable i for $i = 1, \ldots (k-1)$ in line 8–13. The carry-in and non carry-in workload of the higher priority jobs of task τ_i that is exclusively affected by g errors are determined in line 10–11 for all $g = 0, \ldots f$. Aydin in [Ayd07] showed that the time complexity to find the workload of a set of $\hat{N}$ aperiodic tasks is $O(\hat{N} \cdot f)$ where these jobs are affected by exactly f errors. The jobs of a set of sporadic tasks, when arrive as early as possible, can be considered as a set of aperiodic tasks and there are at most $\hat{N}$ jobs within the

Algorithm **FTGS-Test(**Γ, f, m**)**

1. For $k = 1$ to n
2. $\quad C_k^0 = C_k$
3. $\quad$ For $c = (f-1)$ to 0
4. $\quad\quad C_k^{(f-c)} = E_k^{(f-c)} + C_k^{(f-c-1)}$
5. $\quad$ End For
6. End For
7. For $k = (m+1)$ to n
8. $\quad$ For $i = 1$ to $(k-1)$
9. $\quad\quad$ For $g = 0$ to f
10. $\quad\quad\quad$ Find $\texttt{WNC}_i^g(D_k, \xi)$ using Eq. (8.10)
11. $\quad\quad\quad$ Find $\texttt{WCI}_i^g(D_k, \xi)$ using Eq. (8.11)
12. $\quad\quad$ End For
13. $\quad$ End For
14. $\quad$ For $i = 1$ to $(k-1)$
15. $\quad\quad$ For $c = 0$ to f
16. $\quad\quad\quad$ For $g = 0$ to c
17. $\quad\quad\quad\quad$ Find $\texttt{INC}_i^g(D_k, c)$ using Eq. (8.12)
18. $\quad\quad\quad\quad$ Find $\texttt{ICI}_i^g(D_k, c)$ using Eq. (8.13)
19. $\quad\quad\quad\quad$ Find $\texttt{DIFF}_g(\{\tau_i\}, D_k, c)$ using Eq. (8.15)
20. $\quad\quad\quad$ End For
21. $\quad\quad$ End For
22. $\quad$ End For
23. $\quad$ For $c = 0$ to f
24. $\quad\quad$ For $q = 0$ to c
25. $\quad\quad\quad A =$ FindCITasks($\texttt{HP}_k$, q, $m-1$, c)
26. $\quad\quad\quad B = \texttt{HP}_k - A$
27. $\quad\quad\quad I = \texttt{TI}^q(\text{A}, D_k, c) + \texttt{TI}^{c-q}(\text{B}, D_k, c)$
28. $\quad\quad\quad$ If $(\lfloor \frac{I}{m} \rfloor + C_k^{(f-c)} > D_k)$ then
29. $\quad\quad\quad\quad$ Return "False"
30. $\quad\quad\quad$ End if
31. $\quad\quad$ End For
32. $\quad$ End For
33. End For
34. Return "True"

Figure 8.4: *Pseudocode of* **FTGS-Test(**Γ, f, m**)**

scheduling window of each task. Thus, the time complexity to find the value of carry-in and non carry-in workload using Eq. (8.10) and Eq. (8.11) is $O(\hat{N} \cdot g)$ for a particular g and a particular task τ_i. And, thus the time complexity for evaluating Eq. (8.10) and Eq. (8.11) for all $g = 0, 1, \ldots f$ and for all tasks in line 8–13 is $O(n \cdot \hat{N} \cdot f^2)$.

Based on the workload of each of the higher priority tasks, individual interfering workload of each higher priority task in line 14–22 is calculated. Given the values of $C_k^{(f-c)}$ and the workload of each task, the value of individual carry-in or non carry-in interfering workload can be calculated using one addition, one subtraction and one comparison using using Eq. (8.12) and Eq. (8.13), respectively. The difference between the carry-in and non carry-in individual interfering workloads of each task τ_i is determined in line 19. The value of $\texttt{DIFF}_g(\{\tau_i\}, D_k, c)$ using Eq. (8.15) is determined in line 19 using only one subtraction operation. Thus, the time complexity to find the individual carry-in and non carry-in interfering workload and the difference between them for all $c = 0, \ldots f$, for all $i = 1, 2, \ldots (k-1)$ and for all $g = 0, 1, \ldots c$ is $O(n \cdot f^2)$.

When evaluating Eq. (8.26) for task τ_k, one has to consider c errors affecting the higher priority tasks in $\texttt{HP}_k$ and the remaining $(f-c)$ errors affecting task τ_k within an interval of length D_k for $c = 0, \cdots f$. Moreover, for a given a value of c, a total of q errors affecting only the $(m-1)$ higher priority carry-in tasks and $(c-q)$ errors affecting the higher priority non carry-in tasks are considered for $q = 0, \ldots c$. The two nested for loops in line 23–24 consider each possible values of c and q where $0 \leq c \leq f$ and $0 \leq q \leq c$. The $(m-1)$ carry-in tasks are determined by calling `FindCITasks`($\texttt{HP}_k$, q, $m-1$, c) in line 25 for particular values of c and q. The non carry-in tasks from set $\texttt{HP}_k$ are determined in line 26 using one set difference operation. However, it is not needed to perform this set difference operation since algorithm `FindCITasks` in Figure 8.3 can easily determine the set of non carry-in tasks while determining the set of carry-in tasks. Remember that the time complexity to find the $(m-1)$ carry-in tasks using `FindCITasks` is $O(n \cdot f \cdot max\{m, f\})$. Given the carry-in and non carry-in tasks in sets A and B (line 25–26), the total interfering workload of all the tasks in $\texttt{HP}_k = A \cup B$ can be determined using Eq. (8.23).

The total interfering workload of the carry-in tasks and non carry-in tasks respectively in sets A and B are given as $\texttt{TI}^q(\text{A}, D_k, c)$ and $\texttt{TI}^{g-q}(\text{B}, D_k, c)$ using Eq. (8.23). The sum of $\texttt{TI}^q(\text{A}, D_k, c)$ and $\texttt{TI}^{g-q}(\text{B}, D_k, c)$ in line 27 is the total interfering workload of the higher priority tasks that are affected by c errors where q errors affect the carry-in task and $(c-q)$ errors affect the non carry-in tasks. It is not difficult to see that the time complexity to find the values of $\texttt{TI}^q(\text{A}, D_k, c)$ and $\texttt{TI}^{g-q}(\text{B}, D_k, c)$ for a particular c, where $c \leq f$, using Eq. (8.23) is $O(n \cdot f)$.

Evaluating the condition in line 28–30 can be done in constant time. Thus, the time complexity to evaluate the condition in line 28 for *all* possible values of c and q using the loops in line 23–24 is $O(n \cdot f^3 \cdot max\{m, f\})$ when evaluating the schedulability test for task τ_k. By adding the time complexities of all the steps, the time complexity to evaluate `FTGS-Test(`Γ, f, m`)` in Figure 8.4 is $O(n^2 \cdot f^2 \cdot max\{\hat{N}, m \cdot f, f^2\})$ which is pseudo-polynomial in the representation of the task set and fault model.

8.8 Tolerating Processor Failures

In this section, the schedulability test `FTGS-Test(`Γ, f, m`)` is extended in order to determine whether the effect of ρ permanent processor failures can be mitigated using

FTGS algorithm. Remember that FTGS scheduler deals with a processor failure by assuming the task that was executing on the faulty processor has encountered a task error. Once a processor failure is detected, the FTGS scheduler performs the following two actions:

- no task is dispatched to the faulty processor, and
- if any task was executing on the faulty processor, its backup is stored in the ready queue.

The fault model for processor failure considers fail-stop processors and includes simultaneous multiple processor failures. The execution requirement of the recovery operations at any time instant due to processor failures is maximum if all the ρ processors fail simultaneously at that time instant while each of these ρ processors is executing some task. This is the worst-case scenario for ρ processor failures since the backups of total ρ tasks that were executing on the faulty processors simultaneously become ready. And, the multiprocessor platform now has $(m - \rho)$ non-faulty processors. Consequently, there is an interval of length D_{max} in which it is required to consider total $(f + \rho)$ task errors that need to be tolerated using FTGS scheduling on $(m - \rho)$ processors. Note that tolerating both task errors and processor failures using FTGS algorithm requires each task to have $(f + \rho)$ backups. *The extended schedulability test for FTGS algorithm to tolerate both task errors and processor failures is given as follows: apply FTGS-Test(Γ, $f + \rho$, $m - \rho$) to determine whether the answer to this schedulability test is positive or negative.*

Resilience: Given a sporadic task set Γ, the system designer can apply the proposed FTGS-Test(Γ, $f + \rho$, $m - \rho$) for various combinations of the parameters f, m and ρ. An exhaustive approach to judge the resilience of the fault-tolerant system would be to apply the FTGS-Test(Γ, $f + \rho$, $m - \rho$) on all possible triplets (m, f, ρ) where $m \in \{2, 3, \ldots\}$, $f \in \{0, 1, \ldots\}$ and $\rho \in \{1, 2, \ldots m\}$. The system designer can also determine the minimum number of processors required for scheduling an embedded real-time application for some given f and ρ using FTGS algorithm.

Effective Priority Assignment Policy: It is not difficult to see that the schedulability test FTGS-Test(Γ, f, m) is OPA-compatible (i.e., the three OPA-compatibility conditions in page 83 are satisfied). Thus, it can be used to determine effective fixed-priority assignment by applying the combination of multiprocessor extension of Audsley's optimal priority assignment policy. Moreover, this test can also take the advantage of the hybrid priority assignment policy using the separation criterion that is proposed in Chapter 6. In addition, instead of performing deadline-analysis, a response-time based analysis similar to the IA-RT test can be performed (please see chapter 6). These three features (i.e, OPA+HPA+RTA) would result progressively better tests than the proposed FTGS-Test(Γ, f, m) for the FTGS scheduling.

Configuring FTGS for Active Backups: The FTGS scheduling algorithm and the schedulability test FTGS-Test(Γ, f, m) considers passive backups: a backup task

only becomes ready if a task error is detected, otherwise, the backup never executes. However, it is possible to configure FTGS scheduling algorithm to consider active backups as well. Active backups consumes more CPU cycles in comparison to passive backups but provides quick error recovery to the tasks. Such quick error recovery is needed for low-laxity tasks. The basic idea for incorporating active backups is described below.

Consider that there are f' backups of each task τ_i that are active backups where $0 \leq f' \leq f$. Without loss of generality, consider that the first f' backups of each task τ_i are the active backups while the remaining $(f - f')$ backups are passive backups. In such case, the primary and the f' backups of each task become ready whenever a job of the task is released. The priority of the active backups are same as that of the primary. In contrast to complete passive backup, the active backups always execute no matter what happens to the primary or other active backups. If an error is detected after execution of any one of these active backups or the primary, the first passive backup becomes ready for execution. Subsequent error detected in any one of the currently active backups or the primary results in next passive backup to become ready for execution. However, as soon as either the primary or any of the backups of a task completes execution without signaling an error, the other active backups of the task can be terminated. Such backup deallocation utilizes the processors efficiently without sacrificing fault-tolerance.

In order to ensure that all the tasks are schedulable using the combined active-passive backup policy, new schedulability test has to be derived. A preliminary idea is to consider each of the active backups as a different task. Thus, there will be f' additional (pseudo) tasks for each original task τ_i. A new task set is formed by including for each task $\tau_i \in \Gamma$, a task corresponding to the primary and the f' tasks corresponding to the f' active backups. This new task set has total $(n + n \cdot f')$ tasks. If this new task is global FP schedulable (without considering faults), then at most f' task errors can be tolerated within any interval of length D_{max}. To tolerate an additional $(f - f')$ task errors within any time interval of length D_{max}, it is sufficient to show that an additional $(f - f')$ task errors within any time interval of length D_i can be tolerated when considering the schedulability of a lower priority task τ_i. therefore, if this new task set can tolerate $(f - f')$ task errors within any interval of length D_{max}, then all the deadlines are met on m processors while tolerating f tasks errors in any interval of length D_{max}. The FTDM scheduling algorithm proposed in last chapter for uniprocessor fault-tolerant scheduling can also be extended to incorporate active backups.

8.9 Graceful Degradation

The fault tolerant scheduling algorithms FTDM and FTGS proposed respectively in Chapter 7 and Chapter 8 assumes a certain fault model. However, fault-tolerant systems needs to provide correct service even if the errors that occur in the system are not compliant with the fault model. For example, if there are more that f task errors within an interval of length D_{max}, then the proposed schedulability analysis can not ensure that all the deadlines will be met. In such case, upon detection of an error if the recovery operation (i.e., execution of a backup) can not be guaranteed to meet the deadline of the task, the

system should be robust enough such that it provides degraded service in a graceful way. An admission controller in such can decide whether to accept or reject such a recovery request. Three possible alternatives for handling the recovery request are proposed:

- **Direct Rejection:** Simply reject the request without any further consideration.
- **Criticality-Based Eviction:** Evict some low-criticality task from the system to accept the new recovery request.
- **Imprecise Computation:** Accept the new recovery request and execute as much as possible of the corresponding backup without compromising the timeliness of other tasks.

8.9.1 Direct Rejection

If an error is detected and the recovery request can not be accepted, for example by the admission controller of the fault-tolerant scheduling algorithm, then the simple approach is to just rejecting the recovery request. If the system is already highly-loaded, the recovery request is most probably be rejected and in such case the reliability of the system is degraded so as to guarantee schedulability of the other existing tasks.

8.9.2 Criticality-Based Eviction

If an error is detected and the recovery request can not be accepted by the admission controller of the fault-tolerant scheduling algorithm, then *criticality-based eviction* can be employed. In this approach, some already-admitted task, having lower criticality than the criticality of the recovery request, is temporarily terminated and the recovery request is serviced. The termination of the lower-criticality task is temporary in the sense that, when the backup corresponding to the recovery finishes execution, the evicted lower criticality task can be re-admitted into the system. In such case, the lower-criticality task may be unable to execute its jobs that are released while recovery operation is being performed.

By criticality of a task it means the user-perceived importance of the applications tasks in meeting the deadlines. The criticality of the tasks in a task set can be determined independent of the priorities of the tasks [MAM99]. Such criticality-based eviction is applicable for applications in which execution of some jobs of a task can be skipped. In [CB98], scheduling of hard and firm periodic tasks are considered. A firm task can occasionally skip one of its jobs based on some predetermined quality-of-service agreement while the hard periodic task must execute all of its jobs.

Criticality-based scheduling for non-deterministic workloads is addressed by Alvarez and Mossé in [MAM99]. They analyzed the schedulability of a fixed-priority system using a concept called responsiveness [MAM99]. Their analysis is best suited for systems with nondeterministic workload in which recovery operations caused by faults are serviced at different responsiveness levels. By responsiveness level, the authors

mean whether the recovery operation is run in a non-intrusive (without affecting schedulability of other tasks) or intrusive (affecting schedulability of existing tasks) manner. In case of intrusive recovery, timeliness of the less-critical tasks are compromised and the system suffers degraded service. Thus, the eviction of lower-criticality task degrades schedulability performance but provides higher reliability.

Note that, such criticality-based eviction may not work if there is no lower-criticality task to evict in order to accept a recovery request, or if negating the total computation demand of all the lower-criticality tasks from the system is not enough for executing the recovery request. This problem can be addressed using imprecise computation paradigm.

8.9.3 Imprecise Computation

If partial computation of the recovery request is useful, then the recovery request can be accepted into the system even though a complete recovery request can not be serviced due insufficient processing capacity. When the result of a complete execution of a recovery request can not be produced before the deadline, errors (outside the scope of the consider fault model) can be recovered using *imprecise computation* of the backup. Imprecise computation models are considered in [CLL90, LSL+94, MAAMM00] and are appropriate for monotone processes where result produced by a task will have increasingly higher quality if more time is spent in executing the task. Such monotone processes are considered to have a mandatory part and an optional part [LSL+94]. The mandatory part of each task has a hard deadline and must complete its execution before deadline. However, the optional part of a task can be skipped if enough processing power is not available.

The imprecise computational model is applicable if the backup of a faulty task is modeled as a monotone process. Therefore, even if the full execution of the backup can not be completed, the result of the partial computation of the backup can ensure certain quality to the application. Hence, when the admission controller can not guarantee complete execution of a recovery request, the request can still be accepted to the system and imprecise result can be delivered to the application. By considering the recovery request as a monotone process, the imprecise computation technique to serve a recovery request can be seen as providing a balance between schedulability performance and reliability.

It is easy to realize that eviction of a low-criticality task and imprecise computation can be combined so as to offer a solution to the problem where the mandatory part of a task does not have enough time to finish before its hard deadline. In such case evicting a lower criticality task could enable the complete execution of the mandatory part of a highly-critical recovery request.

8.10 Summary

In this chapter, a fault-tolerant multiprocessor scheduling algorithm called FTGS and its corresponding schedulability test for constrained-deadline sporadic tasks are proposed.

This schedulability test enables the system designer to judge the robustness of the system by experimenting with different number of task errors and processor failures. Such sensitivity analysis enables the designers to evaluate off-line the resource requirement and resilience of the fault-tolerant system. The fault model that FTGS algorithm considers is very powerful in the sense that multiple task errors or processor failures are considered to occur at any time, in any task, or even during the execution of recovery operation. No other works have considered such a general fault model for scheduling real-time sporadic tasks on multiprocessors.

The FTGS scheduling considers passive backups: a backup is dispatched after an error is detected. Such passive-backup strategy is good in terms of saving CPU cycles for systems where faults are less likely. Passive backups are also effective for tasks that have enough laxity so that there is enough time in the schedule to execute the backup after an error is detected. However, for low-laxity tasks, passive backups may not be effective to provide fault-tolerance and active backups may be appropriate in such case. However, active backup strategy consumes more energy but provides quick recovery. The system designer can determine for the FTGS algorithm whether only active backups, only passive backups, or a combined approach to be used for the system.

The FTGS scheduling algorithm and its analysis can be extended both for an improved priority assignment policy. The proposed schedulability test for FTGS algorithm is OPA-compatible and can be used to find a fixed-priority ordering of the tasks if the schedulability test is not satisfied for the given fixed-priority ordering of the tasks. Moreover by prudently keeping some tasks and processor separated from the schedulability analysis of a lower priority task, better priority assignment policy based on the HPA scheme can be obtained.

9 Mixed-Criticality Systems

The advent of multicore processors has attracted many safety-critical systems, e.g., automotive and avionics, to consider integrating multiple functionalities on a single, powerful computing platform. Such integration leads to host functionalities with different criticality levels on the same platform. The design of such "mixed-criticality" systems is often subject to certification from one or more certification authorities. Coming up with an effective scheduling policy and its analysis that can guarantee certification of the system at each criticality level, while maximizing the utilization of the processors, is the focus of the research presented in this chapter.

The global, fixed-priority scheduling algorithm for a set of constrained-deadline and mixed-criticality sporadic tasks on multiprocessors is considered. A sufficient schedulability test based on response-time analysis of the proposed algorithm is derived. One of the useful features of the proposed test is that it can be used for systems with more than two criticality levels. In addition, the test can be used to find "effective" fixed-priority ordering of the mixed-criticality tasks based on Audsley's approach. Empirical investigation into the effectiveness of Audsley's priority assignment algorithm using the proposed schedulability test shows significant improvement over other heuristic-based (e.g., deadline-monotonic, criticality-monotonic) priority assignment policies.

9.1 Introduction

Single-chip multiprocessors are viewed as serious contenders for many safety-critical and hard real-time systems to meet the growing demand of computing power. The designers of such systems are considering integrating multiple functionalities on the same

computing platform due to space, weight and power concerns. For example, aviation industry is contemplating "Integrated Modular Avionics" (IMA) to achieve economic advantage by hosting multiple avionics functions on a single platform [ARI]. Similarly, the growing complexity and safety requirements in automotive systems have led to the development of the AUTOSAR framework focusing on composability of components [AUT]. Version R4.0 of AUTOSAR provides the specification for multicore OS architectures.

The functionalities of safety-critical applications, e.g., control and monitoring, are often modeled as a collection of real-time, sporadic tasks having *hard deadlines*. A MC real-time system is the one in which the *criticality levels*, i.e., importance, of different real-time tasks may be different. The design of MC systems is often subject to certification at each criticality level by standard statutory certification authority (CA), for example, by Federal Aviation Authority in the US or the European Aviation Safety Agency in Europe for avionics systems. One of the major challenges in designing MC real-time systems is devising a scheduling strategy that addresses both the "criticality" and "deadline" aspects of the tasks while facilitating certification and efficient resource usage.

In order to certify a MC system as being correct, the CAs make certain assumptions about the worst-case behavior of the system. In this thesis, a particular aspect of the runtime behavior of the system: the WCET of the application tasks is considered. Vestal has pointed out in [Ves07] that *the more confidence one needs in a task execution time bound, the larger and more conservative that bound tends to be in practice*. The CAs become increasingly pessimistic regarding their estimation of the WCET of a piece of code for increasingly higher criticality levels. However, the CA, when certifying the system at some criticality level, is also concerned about the correctness (i.e., meeting the deadlines) of the real-time tasks relevant *only* to that particular criticality level. For example, in order to operate Unmanned Aerial Vehicle (UAV) over civilian airspace, the *flight-critical* functionalities must be certified as "correct" by the CA while the manufacturer needs to ensure the correctness of both *mission-critical* and flight-critical functionalities. Due to such different assumptions and concerns among the CAs and the manufacturers, conventional scheduling strategies addressing both the "criticality" and "deadline" aspects of MC systems are not cost- and resource-efficient. This is illustrated using a contrived example:

Example 9.1. *Consider six constrained-deadline periodic tasks $\tau_1 \ldots \tau_6$ that are to be scheduled on $m = 2$ identical processors based on global* FP *scheduling. Assume that all the tasks are released at time zero and there are only two criticality levels (i.e., dual-criticality system): tasks τ_1 and τ_2 are low-critical tasks while the other tasks $\tau_3 \ldots \tau_6$ are the high-critical tasks.*

The period of each task is 7. The relative deadline of each of the low-critical tasks τ_1 and τ_2 is 4. The relative deadline of each of the high-critical tasks $\tau_3 \ldots \tau_6$ is 7. According to the system designer, the WCET of each task is 2. According to the CA, the WCET of each of the higher-critical[1] tasks $\tau_3 \ldots \tau_6$ is 3. Scheduling the tasks us-

[1]The CA is not concerned about the low-critical tasks and does not specify their execution times.

ing global FP *scheduling requires each of the tasks to have one distinct fixed-priority between priority level 1 (highest) to 6 (lowest).*

If any of the low-critical tasks τ_1 or τ_2 is assigned priority level 5 or 6, then that task misses its deadline even if each of the high critical tasks $\{\tau_3, \ldots \tau_6\}$ actually executes for at most 2 time units (the system designer is not happy with the schedule). If none of the tasks τ_1 and τ_2 is assigned priority level 5 or 6, then **not** *all the high-critical tasks $\{\tau_3, \ldots \tau_6\}$ meet their deadlines when they execute for 3 time units at run-time (the CA is not happy with the schedule). Thus, the system can not be scheduled in a manner that satisfies both the system designer and the CA if traditional global* FP *scheduling is used. However, there is a valid schedule that can satisfy both parties.*

- *Consider that the global* FP *scheduling algorithm is augmented with runtime monitoring support that can monitor the execution time of each job of each task, i.e., can determine how long a job has been executing.*
- *Task τ_3 and τ_4 are assigned the highest priority levels 1 and 2. Task τ_1 and τ_2 are assigned the next two priority levels 3 and 4. And, task τ_5 and τ_6 are assigned the lowest two priority levels 5 and 6.*
- *Note that the hyperperiod of the task set is 7 and within each hyperperiod exactly one job of each task is released. Therefore, if the job of each task is schedulable in the first any hyperperiod, then all the jobs of all the tasks are schedulable.*
- *First, the tasks τ_3 and τ_4 are executed within the hyperperiod since these are the two highest priority tasks and there are two processors.*
- *If any of the two jobs of these two tasks τ_3 and τ_4 does not signal completion of execution after executing for 2 time units (i.e., the assumption of the system designer does not hold), then low-critical tasks τ_1 and τ_2 are dropped from the system. And, each of the jobs of the high-critical tasks $\{\tau_3, \ldots \tau_6\}$ can execute for at most 3 time units within each hyperperiod and can meet their deadlines.*
- *If both jobs of tasks τ_3 and τ_4 signal completion after executing for at most 2 time units, then the two jobs of the low-critical tasks τ_1 and τ_2 are executed for 2 time units and can meet their deadlines. Finally, the two jobs of the high-critical tasks τ_5 and τ_6 can execute for at most 3 time units and can also meet their deadlines.*

So, if the system designer's assumption (that each job execute for 2 time units) hold during runtime, then all tasks meet their deadlines according to global FP *scheduling. If the CA is right (that each high-critical job executes for 3 time units), then all the deadlines are met. Thus, both the CA and the system designers are satisfied.* □

It is evident that the schedulability of the MC task systems in Example 9.1 can not be guaranteed based on traditional global FP scheduling algorithm. No work has proposed scheduling of constrained-deadline MC sporadic tasks on multiprocessors based on the industry-preferred FP scheduling. The only work on multiprocessor scheduling of MC tasks is recently proposed by Li and Baruah in [LB12] considering dynamic

priority and implicit-deadline tasks. The study of FP scheduling algorithms and their analysis on multiprocessors for MC constrained-deadline sporadic tasks is the focus of this chapter.

In this thesis, an implementation scheme of global FP scheduling, called Mixed-criticality Scheduling algorithm on Multiprocessors (MSM), for dispatching a set of mixed-criticality, sporadic tasks on m identical processors is proposed. The proposed algorithm MSM essentially dispatches tasks in accordance to traditional global FP scheduling but has two additional implementation features: (i) the duration of the execution time of each job is *monitored* at run-time in order to detect any transition of the system's behavior to a higher criticality level, and (ii) upon detection of such transition at runtime, some tasks are *dropped* to better utilize the processors without violating the certification requirements. The run-time monitoring support exists in many safety critical-system where the execution time of each job is monitored in order to provide temporal guarantees, fault-tolerance or health monitoring [AB98, CJD91, PMCR08, RRJ92, HS89]. And, this capability is exploited in this thesis for the design and analysis of certification-cognizant multiprocessor FP scheduling of MC systems.

The main contribution in this chapter is the derivation of a sufficient schedulability condition of the MSM algorithm based on response time analysis (RTA) that can be used to guarantee certification at each criticality level. One of the novel features of the proposed schedulability test is that it can be used to find "effective" fixed-priority ordering of the MC tasks based on Audsley's optimal priority assignment (OPA) algorithm [Aud01]. When a MC task set for a given priority ordering does not satisfy the proposed schedulability test, a different priority ordering for which the task set satisfies the schedulability test may be determined using Audsley's algorithm. This is an important feature since the optimal fixed-priority ordering, even for traditional (non-MC) sporadic tasks on multiprocessors, is still unknown. Another useful feature of the proposed test is that it is applicable to systems having more than two criticality levels. This feature is important as many safety-critical systems (i.e., automotive, avionics) that have more than two criticality levels.

The chapter is organized as follows: Section 9.2 presents the system model and the MSM algorithm. The basic framework for the schedulability analysis of the MSM algorithm is presented in Section 9.3. The schedulability analysis of the MSM algorithm for dual-critical systems is presented in Sections 9.4–9.5. Then, the schedulability analysis for arbitrary number of criticality levels is presented in Section 9.6. Empirical investigation into the proposed schedulability test and priority assignment policy is presented in Section 9.7. The related works are presented in Section 9.8 before concluding the chapter in Section 9.9.

9.2 System Model and The Scheduler

The preemptive scheduling of MC sporadic task systems on m identical processors is considered. A MC sporadic task system Γ consists of n mixed-criticality sporadic tasks $\tau_1, \ldots, \tau_n$ having $\mathcal{L}$ distinct criticality levels. Each task τ_i is characterized by a 4-tuple

(L_i, D_i, T_i, C_i), where

- $L_i \in \{1, 2, \ldots \mathcal{L}\}$ is the criticality level of the task where $\mathcal{L}$ is the highest criticality level in the system.
- $T_i \in \mathbb{N}^+$ is the minimum inter-arrival time of the jobs (also, called period) of the task.
- $D_i \in \mathbb{N}^+$ is the relative deadline such that $D_i \leq T_i$.
- C_i is a vector $< C_i^1, C_i^2, \ldots C_i^{\mathcal{L}} >$ that represents the worst-case execution times of task τ_i at different criticality levels. The WCET of task τ_i at criticality level ℓ is equal to C_i^ℓ.

The WCET of a piece of code is generally an upper bound on the true WCET and the more confidence one needs in estimating the WCET of a piece of code, the more pessimistic this upper bound tends to be. Therefore, different values for WCET of a piece of code can be determined based on the level of confidence one needs in estimating that WCET. To that end, it is assumed that $C_i^\ell \leq C_i^{(\ell+1)}$ for each task $\tau_i \in \Gamma$.

The set of all the *higher priority* tasks of task τ_i is denoted by $\mathtt{HP}_i$. The set of higher-priority but *lower-critical* tasks of task τ_i is denoted by `hpL(i)`. Similarly, the set of higher-priority and *higher/equal-critical* tasks of task τ_i is denoted by `hpH(i)`. Note that, $\mathtt{HP}_i = \mathtt{hpL(i)} \cup \mathtt{hpH(i)}$.

Behavior: A `MC` sporadic task system shows different behavior during different run of the system since different jobs may be released at different time instant and may have different execution times. The system is said to have exhibited *ℓ-criticality behavior* if no job of *any* task τ_i executes for more than C_i^ℓ time units, for some *minimum* ℓ, where $1 \leq \ell \leq \mathcal{L}$. If no such ℓ between 1 and $\mathcal{L}$ exists, then the behavior of the system is *erroneous*.

Correctness: A `MC` system is certified as *correct* if and only if the system is *schedulable* at each criticality level. A `MC` task system is *schedulable at criticality level ℓ* using algorithm $\mathcal{A}$ if and only if the jobs of each task τ_i, satisfying $L_i \geq \ell$, complete by their deadlines for all ℓ-criticality behavior of the system when scheduled using $\mathcal{A}$.

The `MSM` algorithm. The `MSM` algorithm for dispatching the jobs of the `MC` tasks works as follows:

- There is a criticality level indicator ℓ, initialized to the lowest criticality level, $\ell \leftarrow 1$.
- While ($\ell \leq \mathcal{L}$), at each time-instant, the ready jobs of at most m highest-priority tasks with criticality level greater than or equal to ℓ are dispatched for execution on m processors; and
  - if a currently executing job of any task τ_i has executed C_i^ℓ time units without signaling completion, then $\ell \leftarrow (\ell + 1)$.

Algorithm MSM works exactly same as traditional global FP scheduling except that run-time monitoring of the execution of each job is employed to detect the switch from ℓ-criticality to $(\ell+1)$-criticality behavior of the system. And, according to the definition of "correctness" the jobs of ℓ-critical tasks need not be dispatched (hence, dropped by MSM algorithm) as soon as the system switches to $(\ell+1)$-criticality behavior. The system switches from ℓ to $(\ell+1)$-criticality behavior if some job does not signal completion after executing for its ℓ-criticality execution time.

The main objective in this chapter is to derive a schedulability test of the MSM algorithm. Section 9.3 presents the framework for the schedulability analysis of the MSM algorithm. The schedulability analysis is first presented for dual-criticality[2] systems: Section 9.4 and Section 9.5 present the response time analysis considering the LO and HI criticality behavior of the system, respectively. The schedulability analysis for more than two criticality levels is presented in Section 9.6.

9.3 Schedulability Analysis: an Overview

In this section, an overview of the schedulability analysis of the MSM algorithm is presented. To guarantee certification of MC system at criticality level ℓ, each task τ_i satisfying $L_i \geq \ell$ must be schedulable during all ℓ-criticality behavior of the system. A sufficient schedulability test of the MSM algorithm based on response time analysis is derived in this chapter.

The response time of task τ_i is denoted by R_i^ℓ considering the ℓ-criticality behavior of the system. To derive R_i^ℓ, the schedulability analysis of a generic job of task τ_i in an interval of length t, called the "problem window" of task τ_i, is considered. The response time of task τ_i is derived by computing the *workload*, *interfering workload*, *total interfering workload* and *interference* of the higher priority tasks within the problem window.[3]

The CI and NC **workloads** of each higher priority task $\tau_k \in \mathsf{HP}_i$ within the problem window of length t are determined. Whether a task τ_i should be considered as a CI task or a NC task is determined later. The CI and NC **interfering workloads** of each higher priority task $\tau_k \in \mathsf{HP}_i$ are determined based on the upper bound on the CI and NC workloads of task τ_k within the problem window, respectively.

It is proved in [GSYY09] that there are at most $(m-1)$ carry-in tasks in the problem window of any lower priority task for global FP scheduling of constrained-deadline sporadic tasks. Since the MSM algorithm essentially dispatches the MC tasks based on global FP scheduling policy, limiting the number of CI tasks to $(m-1)$ is also applicable for the schedulability analysis of MSM algorithm. The **total interfering workload** is calculated by adding the CI interfering workloads of $(m-1)$ carry-in tasks and the NC interfering workloads of the remaining higher priority tasks. The $(m-1)$ carry-in tasks from set HP_i are selected such that the total interfering workload is maximized.

[2] The criticality levels 1 and 2 are denoted by "LO" and "HI".

[3] The terms (i.e., workload, interfering workload, total interfering workload and interference) are formally defined in Section 6.2 (see page 81).

Finally, the **interference** due to the tasks in HP_i in the problem window of task τ_i is calculated based on total interfering workload of the tasks in HP_i.

Once the interference of the higher priority tasks within a problem window considering the ℓ-criticality behavior of the system is calculated, the response time R_i^ℓ of task τ_i is given as a recurrence that can be solved using fixed-point iteration technique. This response-time test is derived by assuming some *given* fixed-priority ordering of the tasks. However, determining a "good" fixed-priority ordering of the MC tasks is as important as deriving a schedulability test. This is because if a task set does not pass the schedulability test for a given priority ordering, then a priority ordering for which the task set passes the schedulability test can avoid unnecessary upgrade of hardware or re-specification of software. The Audsley's OPA algorithm [Aud01] combined with the proposed (response-time based) schedulability test in this chapter will be applied to find an effective fixed-priority ordering of the MC tasks.

9.3.1 Dual-Criticality Systems

A dual-criticality system exhibits either LO or HI criticality behavior. The response time R_i^{LO} and R_i^{HI} of task τ_i will be derived for the LO and HI-criticality behavior of the dual-criticality system, respectively. The following Lemma is used in Sections 9.4–9.5.

Lemma 9.1. *If task τ_j meets all its deadlines during all correct behaviors of a dual-criticality system, then*

$$R_j^{LO} \leq \zeta_j$$

$$\textit{where, } \zeta_j = \begin{cases} D_j - (C_j^{HI} - C_j^{LO}) & \textit{if } L_j = \mathsf{HI} \\ D_j & \textit{if } L_j = \mathsf{LO} \end{cases} \tag{9.1}$$

Proof. Consider a job of task τ_j that finishes C_j^{LO} units of execution exactly R_j^{LO} time units after its release time without signaling completion. If $R_j^{\mathsf{LO}} > D_j - (C_j^{\mathsf{HI}} - C_j^{\mathsf{LO}})$ and $L_j = \mathsf{HI}$, then this job can not complete additional $(C_j^{\mathsf{HI}} - C_j^{\mathsf{LO}})$ units of execution before its deadline during the HI-criticality behavior of the system. Therefore, if task τ_j meets its deadline in all correct behavior of the system and $L_j = \mathsf{HI}$, then $R_j^{\mathsf{LO}} \leq D_j - (C_j^{\mathsf{HI}} - C_j^{\mathsf{LO}})$. And obviously, if $L_j = \mathsf{LO}$, then $R_j^{\mathsf{LO}} \leq D_j$ for all correct behavior of the system. □

According to Lemma 9.1, a job of task τ_j that is released at time r must finish C_j^{LO} units of execution by time $(r + \zeta_j)$ in all LO-criticality behaviors. Lemma 9.1 essentially captures the "true" relative deadline of task τ_j for the LO-criticality behavior of the system. The relative deadline of task τ_j when analyzing the LO criticality behavior of the system is denoted by ζ_j. The relative deadline of task τ_j during the HI criticality behavior of the system is still equal to D_j.

9.4 RTA Procedure at LO Criticality Level

A dual-criticality system is schedulable at the LO criticality level if and only if *each* task $\tau_i \in \Gamma$ meet their deadlines for all LO-criticality behaviors of the system. In this section, the response time R_i^{LO} of task τ_i considering the LO-criticality behavior of the system is derived. According to the MSM algorithm and Lemma 9.1, the execution of any task $\tau_j \in (\mathsf{HP}_i \cup \{\tau_i\})$ during the LO-criticality behavior of the system is equivalent to traditional global FP scheduling of (non-MC) sporadic task τ_j with parameters $(C_j^{\mathsf{LO}}, \zeta_j, T_j)$. In such case, the response time R_i^{LO} of task τ_i can be determined using standard RTA technique proposed for (non-MC) sporadic task systems, for example, using the test proposed by Guan et al. in [GSYY09]. However, the test proposed by Guan et al. in [GSYY09] is OPA-incompatible [DB11b], i.e., it can not be used to find effective fixed-priority ordering of the tasks based on Audsley's approach. Now in subsection 9.4.1 a new, OPA-compatible response time test that can be used to determine the schedulability of task τ_i is presented.

9.4.1 New RTA for Sporadic Task Systems

The response time R_i^{LO} of task τ_i is determined by calculating the workload, interfering workload, total interfering workload and interference of the higher priority tasks within the problem window of task τ_i.

Workload. The CI and NC workloads of each higher priority task $\tau_k \in \mathsf{HP}_i$ within the problem window of task τ_i need to be computed. The upper bound on the workload of task $\tau_k \in \mathsf{HP}_i$ within any interval of length t is denoted by $\mathsf{W}_k^{\mathsf{NC}}(t)$ and $\mathsf{W}_k^{\mathsf{CI}}(t)$ whenever τ_k is a NC task and CI task, respectively. Since each job of task τ_k executes at most C_k^{LO} time units during the LO-criticality behavior, the NC workload $\mathsf{W}_k^{\mathsf{NC}}(t)$ of task τ_k is given (based on [GSYY09]) as follows:

$$\mathsf{W}_k^{\mathsf{NC}}(t) = \lfloor t/T_k \rfloor \cdot C_k^{\mathsf{LO}} + min(C_k^{\mathsf{LO}}, t - \lfloor t/T_k \rfloor \cdot T_k) \tag{9.2}$$

Guan et al. in [GSYY09] also proposed a novel technique for estimating the CI workload of task τ_k within the problem window of τ_i. However, the CI workload computation of task τ_k, according to [GSYY09], requires to know the response time of task τ_k which in turn requires to know the *relative priority ordering* of the higher priority tasks in HP_i. This is because without knowing the relative priority ordering of the tasks in HP_i it is not possible to determine the response time of task $\tau_k \in \mathsf{HP}_i$. Such dependency on the relative priority ordering of the higher priority tasks needs to be avoided to derive an OPA-compatible [Aud01, DB11b] schedulability test (the first condition in page 83 for being a test OPA-compatible is not satisfied). This problem is circumvented by using the upper bound on the response time R_k^{LO} of task τ_k according to Lemma 9.1. The value of CI workload $\mathsf{W}_k^{\mathsf{CI}}(t)$ of task τ_k is given as follows:

$$\mathsf{W}_k^{\mathsf{CI}}(t) = A_t^k \cdot C_k^{\mathsf{LO}} + min(C_k^{\mathsf{LO}}, t + \zeta_k - C_k^{\mathsf{LO}} - A_t^k \cdot T_k) \tag{9.3}$$

where $A_t^k = \lfloor (t + \zeta_k - C_k^{\mathsf{LO}})/T_k \rfloor$ and ζ_k is defined in Eq. (9.1). Note that if R_k^{LO} is used in place of ζ_k in Eq. (9.3), then Eq. (9.3) calculates the same CI workload as in [GSYY09]. However, in order to make the proposed test OPA-compatible, an upper bound on R_k^{LO} (according to Lemma 9.1) is used in Eq. (9.3). It is easy to see that the NC and CI workloads calculation in Eq. (9.2) and Eq. (9.3) do not require to know the relative priority ordering of the tasks in HP_i.

Interfering Workload: The upper bounds on the interfering workload of task τ_k on any job of task τ_i within the problem window of length t are denoted by $\mathtt{I}_{k,i}^{\mathsf{CI}}(t)$ and $\mathtt{I}_{k,i}^{\mathsf{NC}}(t)$ whenever τ_k is a CI task and NC task, respectively. It is pointed out in [BC07, BCL09] that if a job of task τ with execution time C and relative deadline D suffers enough interference to miss its deadline, then it is sufficient to consider the interfering workload of a higher priority task limited to at most $(D - C + 1)$. Therefore, $\mathtt{I}_{k,i}^{\mathsf{CI}}(t)$ and $\mathtt{I}_{k,i}^{\mathsf{NC}}(t)$ are given as follows:

$$\mathtt{I}_{k,i}^{\mathsf{CI}}(t) = min(\mathtt{W}_k^{\mathsf{CI}}(t), t - C_i^{\mathsf{LO}} + 1) \tag{9.4}$$

$$\mathtt{I}_{k,i}^{\mathsf{NC}}(t) = min(\mathtt{W}_k^{\mathsf{NC}}(t), t - C_i^{\mathsf{LO}} + 1) \tag{9.5}$$

The difference between the CI and NC interfering workload of task τ_k within the problem window of length t is denoted by $\mathtt{I}_{k,i}^{\mathsf{DIFF}}(t)$ such that:

$$\mathtt{I}_{k,i}^{\mathsf{DIFF}}(t) = \mathtt{I}_{k,i}^{\mathsf{CI}}(t) - \mathtt{I}_{k,i}^{\mathsf{NC}}(t)$$

Total Interfering Workload. The upper bound on total interfering workload due to all the tasks in set HP_i is denoted by $\mathtt{I}_i(t)$. The value of $\mathtt{I}_i(t)$ is calculated as follows:

$$\mathtt{I}_i(t) = \sum_{\tau_k \in \mathsf{HP}_i} \mathtt{I}_{k,i}^{\mathsf{NC}}(t) + \sum_{\tau_k \in Max(\mathsf{HP}_i, m-1)} \mathtt{I}_{k,i}^{\mathsf{DIFF}}(t) \tag{9.6}$$

where $Max(\mathsf{HP}_i, m-1)$ is the set of $(m-1)$ tasks from set HP_i that have the largest values of $\mathtt{I}_{k,i}^{\mathsf{DIFF}}(t)$.

Interference. The term interference is an integer and all the m processors are busy executing tasks from HP_i while task τ_i is interfered. Thus, an upper bound on interference due to the tasks in HP_i on any job of task τ_i within the problem window of length t is $\lfloor \mathtt{I}_i(t)/m \rfloor$.

The Response Time Test. The response time R_i^{LO} of task τ_i for the LO criticality behavior of the system is given as follows:

$$R_i^{\mathsf{LO}} \leftarrow C_i^{\mathsf{LO}} + \left\lfloor \frac{\mathtt{I}_i(R_i^{\mathsf{LO}})}{m} \right\rfloor \tag{9.7}$$

This can be solved by searching iteratively the least fixed point starting with $R_i^{\mathsf{LO}} = C_i^{\mathsf{LO}}$

for the right-hand side of Eq. (9.7). If $R_i^{\mathsf{LO}} > \zeta_i$, then the task τ_i misses its deadline. When certifying a system at LO criticality level, Eq. (9.7) can be used to determine whether task $\tau_i \in \Gamma$ meets its deadline during all the LO-criticality behavior of the system. Note that Eq. (9.7) can also be used to determine the schedulability of traditional, non-MC, sporadic tasks. The test in Eq. (9.7) does not depend on the relative priority ordering of the higher priority tasks; hence, is OPA-compatible.

An Example: Consider the following dual-criticality task set in Table 9.1 comprised of $n = 3$ tasks to be scheduled using MSM algorithm on $m = 2$ processors.

τ_i	L_i	C_i^{LO}	C_i^{HI}	D_i	T_i	ζ_i
τ_1	HI	1	2	3	4	2
τ_2	LO	1	–	2	3	2
τ_3	HI	2	3	3	4	2

Table 9.1: *An example task set*

Assume that task τ_1 is the *lowest* priority task. The aim is to calculate R_1^{LO} to determine if τ_1 is MSM-schedulable during all LO-criticality behaviors. Note that the other two higher priority tasks τ_2 and τ_3 are trivially schedulable since $m = 2$.

Calculating R_1^{LO}: The response time R_1^{LO} of task τ_1 is calculated in the table below. The first column represents the length of the problem window; initially, set to $R_1^{\mathsf{LO}} = C_1^{\mathsf{LO}} = 1$. The second column presents (based on Eq. (9.6)) the total interfering workload of the higher priority tasks τ_2 and τ_3 for the length of the problem window given in the first column. Finally, the right hand side of Eq. (9.7), i.e., new value of R_1^{LO}, is evaluated and presented in the third column.

R_1^{LO}(problem window)	$\mathtt{I}_1(R_1^{\mathsf{LO}})$	$R_1^{\mathsf{LO}} \leftarrow C_1^{\mathsf{LO}} + \lfloor \frac{\mathtt{I}_1(R_1^{\mathsf{LO}})}{2} \rfloor$
$C_1^{\mathsf{LO}} = 1$	2	$1 + \lfloor \frac{2}{2} \rfloor = 2$
2	3	$1 + \lfloor \frac{3}{2} \rfloor = 2$

Since the values of R_1^{LO} in the third column for the first two iterations are the same, the RTA procedure converges and $R_1^{\mathsf{LO}} = 2$. Since $R_1^{\mathsf{LO}} = 2 \leq \zeta_1 = 2 \leq D_1 = 3$, the deadline of task τ_1 is met for all LO-criticality behaviors. □

9.5 RTA Procedure at HI Criticality Level

A dual-criticality system is schedulable at the HI criticality level if and only if it is true that each HI-critical task in Γ meets their deadlines for all HI-criticality behaviors of the system. In this section, the response time R_i^{HI} of task τ_i considering the HI-criticality behavior of the system is derived.

In order to derive the response time $R_i^{\mathtt{HI}}$ of a HI-critical task τ_i, the schedulability analysis of a generic job J_i^x of task τ_i within the problem window $[r_i^x, r_i^x + t)$ of length t is considered. Assume s be the time instant relative to the release time of job J_i^x at which the system switches from LO to HI criticality behavior (as is given in Figure 9.1).

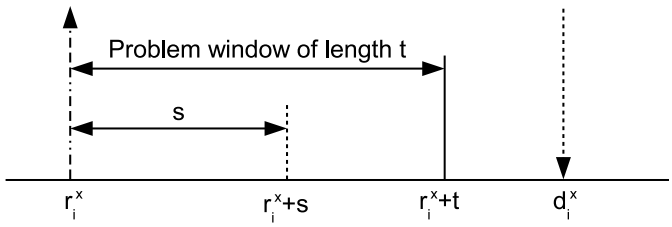

Figure 9.1: *The problem window of length t*

If $s > R_i^{\mathtt{LO}}$, then the system exhibits LO-criticality behavior before $(r_i^x + s)$ and the job J_i^x must have completed before $(r_i^x + s)$ because $(r_i^x + s) > (r_i^x + R_i^{\mathtt{LO}})$. Since the aim is to determine the response time of task τ_i for the HI-criticality behavior of the system, it is sufficient to consider $0 \leq s \leq R_i^{\mathtt{LO}}$ to compute $R_i^{\mathtt{HI}}$.

The response time of task τ_i (i.e., the response time of the generic job J_i^x) for a given value of s is denoted by $R_{i,s}^{\mathtt{HI}}$. The response time $R_i^{\mathtt{HI}}$ is the largest $R_{i,s}^{\mathtt{HI}}$ for some $s, 0 \leq s \leq R_i^{\mathtt{LO}}$. The value of $R_{i,s}^{\mathtt{HI}}$ is calculated based on the workload, interfering workload, total interfering workload and interference of each higher-priority task $\tau_k \in \mathtt{HP}_i$ where $\mathtt{HP}_i = (\mathtt{hpL(i)} \cup \mathtt{hpH(i)})$.

The NC and CI workloads of the higher priority task $\tau_k \in \mathtt{hpL(i)}$ are respectively denoted by $\mathtt{WL}_k^{\mathtt{NC}}(s,t)$ and $\mathtt{WL}_k^{\mathtt{CI}}(s,t)$ such that the system switches from LO to HI criticality behavior at time s relative to the beginning of the problem window of length t. Similarly, $\mathtt{WH}_k^{\mathtt{NC}}(s,t)$ and $\mathtt{WH}_k^{\mathtt{CI}}(s,t)$ denote the NC and CI workloads of task $\tau_k \in \mathtt{hpH(i)}$, respectively.

The remainder of this section is organized as follows. First, the NC and CI workloads of task $\tau_k \in \mathtt{hpL(i)}$ are derived in subsection 9.5.1. Second, the NC and CI workloads of task $\tau_k \in \mathtt{hpH(i)}$ are derived in subsection 9.5.2. Then, the interfering workload, total interfering workload and interference of the tasks in $\mathtt{HP}_i$ are calculated, and finally, a recurrence for $R_{i,s}^{\mathtt{HI}}$ is derived in subsection 9.5.3.

9.5.1 Workload of $\tau_k \in$ hpL(i) within $[r_i^x, r_i^x + t)$

In this subsection, the NC and CI workloads of a LO-critical task $\tau_k \in \mathtt{hpL(i)}$ within the problem window $[r_i^x, r_i^x + t)$ are calculated. According to the MSM algorithm, the LO-critical task τ_k is not dispatched after the criticality-switch at $(r_i^x + s)$. The WCET of task τ_k is $C_k^{\mathtt{LO}}$ since task τ_k executes only during the LO-criticality behavior of the system. The execution of the LO-critical task τ_k in $[r_i^x, r_i^x + s)$ is equivalent to the execution of traditional (non-MC) sporadic task with parameters $(C_k^{\mathtt{LO}}, \zeta_k = D_k, T_k)$. In such case,

$\mathtt{WL}_k^{\mathtt{NC}}(s,t)$ and $\mathtt{WL}_k^{\mathtt{CI}}(s,t)$ are given as follows:

$$\mathtt{WL}_k^{\mathtt{NC}}(s,t) \ = \mathtt{W}_k^{\mathtt{NC}}(s) \tag{9.8}$$

$$\mathtt{WL}_k^{\mathtt{CI}}(s,t) \ = \mathtt{W}_k^{\mathtt{CI}}(s) \tag{9.9}$$

where $\mathtt{W}_k^{\mathtt{NC}}(s)$ and $\mathtt{W}_k^{\mathtt{CI}}(s)$ are given in Eq. (9.2) and Eq. (9.3), respectively.

9.5.2 Workload of $\tau_k \in$ hpH(i) within $[r_i^x, r_i^x + t)$

In this subsection, the NC and CI workloads of a HI-critical task $\tau_k \in$ hpH(i) within the problem window $[r_i^x, r_i^x + t)$ are calculated.

Calculating $\mathtt{WH}_k^{\mathtt{NC}}(s,t)$. The NC workload $\mathtt{WH}_k^{\mathtt{NC}}(s,t)$ of task $\tau_k \in$ hpH(i) within the problem window $[r_i^x, r_i^x + t)$ is calculated according to the releases of the jobs of task τ_k as follows: one job of task τ_k is released at time instant r_i^x and subsequent jobs of task τ_k are released as early as possible. The jobs of task τ_k execute as early as possible within the problem window (as given in Figure 9.2).

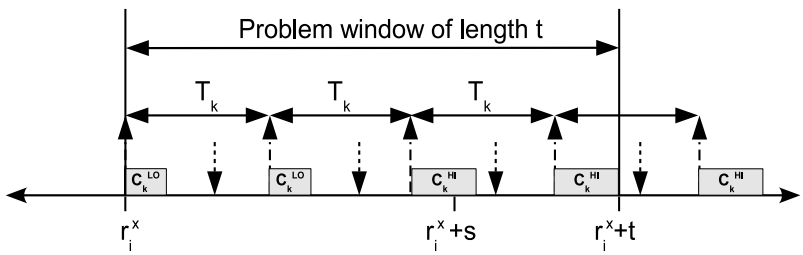

Figure 9.2: *The NC workload of task $\tau_k \in$ hpH(i) within an interval of length t. Note that the criticality changes at time instant $(r_i^x + s)$.*

If each job of task τ_k executes for $C_k^{\mathtt{HI}}$ time units within $[r_i^x, r_i^x + t)$, then the workload of task τ_k within $[r_i^x, r_i^x + t)$, denoted by $W^{upper}(t)$, is given as follows:

$$W^{upper}(t) = \lfloor t/T_k \rfloor \cdot C_k^{\mathtt{HI}} + min(C_k^{\mathtt{HI}}, t - \lfloor t/T_k \rfloor \cdot T_k) \tag{9.10}$$

However, each of at least $\lfloor s/T_k \rfloor$ jobs of task τ_k executes for at most $C_k^{\mathtt{LO}}$ time units within $[r_i^x, r_i^x + s)$. The value of NC workload $\mathtt{WH}_k^{\mathtt{NC}}(s,t)$ is given as follows[4]:

$$\mathtt{WH}_k^{\mathtt{NC}}(s,t) \ = W^{upper}(t) - \lfloor s/T_k \rfloor \cdot (C_k^{\mathtt{HI}} - C_k^{\mathtt{LO}}) \tag{9.11}$$

[4]The job of task τ_k that is released at time $(r_i^x + \lfloor s/T_k \rfloor \cdot T_k)$ executes for at most $C_k^{\mathtt{LO}}$ time units if $(\lfloor s/T_k \rfloor \cdot T_k + R_k^{\mathtt{LO}}) < s$; otherwise, it executes for at most $C_k^{\mathtt{HI}}$ time units. For ease of presentation, this job is assumed to execute for $C_k^{\mathtt{HI}}$ time units.

Calculating $\mathtt{WH}_k^{\mathtt{CI}}(s,t)$. The CI workload $\mathtt{WH}_k^{\mathtt{CI}}(s,t)$ of task $\tau_k \in$ hpH(i) within a problem window of length t is calculated by considering a particular release pattern, called the ***reference pattern***, of the jobs of task τ_k within $[r_i^x, r_i^x + t)$. The reference pattern is defined considering releases of the jobs of task τ_k within $[r_i^x, r_i^x + t)$ as follows (see Figure 9.3):

- one job of task τ_k is released at time $(r_i^x + t - C_k^{\mathtt{HI}})$ and other jobs of τ_k are released as close as possible (periodically) to the job released at $(r_i^x + t - C_k^{\mathtt{HI}})$; and
- the jobs of task τ_k that are released before time instant $(r_i^x + t - C_k^{\mathtt{HI}})$ execute as *late* as possible and the jobs of task τ_k that are released at or after time instant $(r_i^x + t - C_k^{\mathtt{HI}})$ execute as *early* as possible.

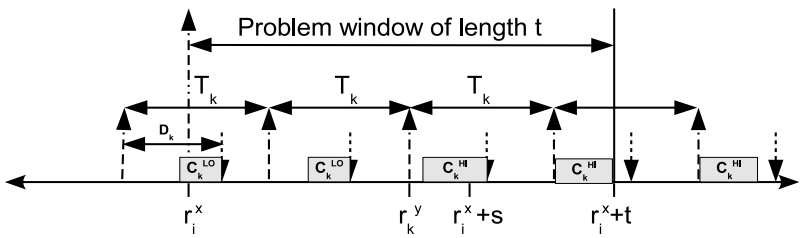

Figure 9.3: *The reference pattern. The criticality-switch occurs at $(r_i^x + s)$ within the interval $[r_i^x, r_i^x + t)$.*

Based on the reference pattern, the value of CI workload $\mathtt{WH}_k^{\mathtt{CI}}(s,t)$ is calculated in two steps as follows:

- **STEP1:** The workload of task τ_k within $[r_i^x, r_i^x + t)$ for the reference pattern in Figure 9.3 is calculated. The workload of task τ_k within the problem window for the reference pattern is denoted by $\mathtt{P}_k(s,t)$.
- **STEP2:** By considering all possible leftward or rightward shifts of the problem window in the reference pattern, the ***maximum net increase*** in workload within the shifted window in comparison to the workload calculated in Step 1 is determined.

The sum of the two workload factors in Step 1 and Step 2 is the value of $\mathtt{WH}_k^{\mathtt{CI}}(s,t)$. The details of calculating the workloads for Step 1 and Step 2 are now presented.

STEP 1 (workload of τ_k in the reference pattern): In this step, the workload $\mathtt{P}_k(s,t)$ of the jobs of task τ_k for the reference pattern in Figure 9.3 is computed. Consider the job J_k^y that satisfies the following condition in the reference pattern:

$$r_k^y \leq (r_i^x + s) < r_k^{(y+1)} \tag{9.12}$$

According to Eq. (9.12), the criticality-switch at $(r_i^x + s)$ occurs at or after the release time of job J_k^y but prior to the release of job $J_k^{(y+1)}$. It is assumed that job J_k^y executes for $C_k^{\mathtt{HI}}$ time units[5]. Any job of task τ_k that is released before and after the release of J_k^y executes for at most $C_k^{\mathtt{LO}}$ and $C_k^{\mathtt{HI}}$ time units in the reference pattern, respectively.

Given the values of t and s, the time instant when job J_k^y is released relative to the time instant r_i^x can be precisely determined. Since job J_k^y satisfies Eq. (9.12), the release time r_k^y of job J_k^y is:

$$r_k^y = (r_i^x + t) - C_k^{\mathtt{HI}} - \mathcal{N}_{t,s}^k \cdot T_k \tag{9.13}$$

where $\mathcal{N}_{t,s}^k$ is number of jobs of task τ_k that are released in $[r_k^y, r_i^x + t - C_k^{\mathtt{HI}})$ in the reference pattern; and $\mathcal{N}_{t,s}^k$ is given as follows:

$$\mathcal{N}_{t,s}^k = \lceil (max\{0, t - C_k^{\mathtt{HI}} - s\})/T_k \rceil \tag{9.14}$$

In other words, the job J_k^y is released $(t - C_k^{\mathtt{HI}} - \mathcal{N}_{t,s}^k \cdot T_k)$ time units apart from the beginning of the problem window. Since τ_k is a $\mathtt{HI}$-critical task, i.e., $L_k = \mathtt{HI}$, the response time of a job of task τ_k that executes for at most $C_k^{\mathtt{LO}}$ time units is upper bounded by $\zeta_k = D_k - (C_k^{\mathtt{HI}} - C_k^{\mathtt{LO}})$ according to Lemma 9.1. In other words, each of the jobs released before r_k^y completes its execution at least $(D_k - \zeta_k) = (C_k^{\mathtt{HI}} - C_k^{\mathtt{LO}})$ time units earlier than its deadline. Based on this observation, the specific reference pattern is depicted in Figure 9.4.

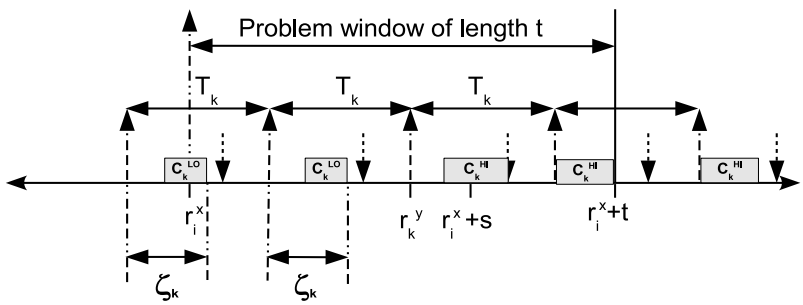

Figure 9.4: *The reference pattern. Each job released before r_k^y finishes $(D_k - \zeta_k)$ time units earlier than its deadline in the reference pattern.*

Observe that no job of task τ_k in Figure 9.4 can execute in $[r_k^y - (T_k - \zeta_k), r_k^y)$ since the job $J_k^{(y-1)}$ completes its execution at or before time instant $r_k^y - (T_k - \zeta_k)$. Thus, the workload of task τ_k within $[r_i^x, r_k^y)$ is in fact the workload of task τ_k within

[5]In fact, job J_k^y executes for $C_k^{\mathtt{LO}}$ time units if $(r_k^y + R_k^{\mathtt{LO}}) < (r_i^x + s)$; otherwise, J_k^y executes for at most $C_k^{\mathtt{HI}}$ time units. For ease of presentation, job J_k^y is assumed to execute for $C_k^{\mathtt{HI}}$ time units.

$[r_i^x, r_k^y - (T_k - \zeta_k))$. The length of the interval $[r_i^x, r_k^y - (T_k - \zeta_k))$ is denoted by Q such that

$$Q = max\{0, t - C_k^{\mathtt{HI}} - \mathcal{N}_{t,s}^k \cdot T_k - (T_k - \zeta_k)\} \tag{9.15}$$

The workload of task τ_k in the reference pattern is calculated considering two cases: Case(A) $Q = 0$, and Case(B) $Q > 0$.

Case (A) ($Q = 0$): For this case, each of the jobs of task τ_k executes for at most $C_k^{\mathtt{HI}}$ time units within the *entire* problem window $[r_i^x, r_i^x + t)$ since $r_i^x \geq 0 \geq t - C_k^{\mathtt{HI}} - \mathcal{N}_{t,s}^k \cdot T_k - (T_k - \zeta_k)$. In other words, the execution of task τ_k is equivalent to the execution of traditional (non-MC) sporadic task τ_k with parameters $(C_k^{\mathtt{HI}}, D_k, T_k)$ in the reference pattern. Based on the work by Bertogna and Cirinei in [BC07] for traditional sporadic tasks, the workload $\mathsf{P}_k(s,t)$ of task τ_k with parameters $(C_k^{\mathtt{HI}}, D_k, T_k)$ in any interval of length t is given as follows:

$$\mathsf{P}_k(s,t) = B_t^k \cdot C_k^{\mathtt{HI}} + min\{C_k^{\mathtt{HI}}, t + D_k - C_k^{\mathtt{HI}} - B_t^k \cdot T_k\} \tag{9.16}$$

where $B_t^k = \lfloor (t + D_k - C_k^{\mathtt{HI}})/T_k \rfloor$.

Case (B) ($Q > 0$): According to Eq. (9.15) for this case, $t > C_k^{\mathtt{HI}} + \mathcal{N}_{t,s}^k \cdot T_k + (T_k - \zeta_k)$. And, according to Eq. (9.13), $r_k^y > r_i^x$ whenever $t > C_k^{\mathtt{HI}} + \mathcal{N}_{t,s}^k \cdot T_k + (T_k - \zeta_k)$. Therefore, job J_k^y is not the *carry-in* job because J_k^y is not released before r_i^x. The workload $\mathsf{P}_k(s,t)$ of task τ_k within $[r_i^x, r_i^x + t)$ for the reference pattern in Figure 9.4 is determined by adding the workload of task τ_k in $[r_i^x, r_k^y)$ and $[r_k^y, r_i^x + t)$.

Remember that the workload of task τ_k within $[r_i^x, r_k^y)$ in Figure 9.4 is in fact the workload of task τ_k within $[r_i^x, r_k^y - (T_k - \zeta_k))$. By viewing the schedule in Figure 9.4 (backward in time), starting from $r_k^y - (T_k - \zeta_k)$ to r_i^x, it is evident that the workload of task τ_k in $[r_i^x, r_k^y - (T_k - \zeta_k))$ is equal to the NC workload of traditional (non-MC) sporadic task τ_k with parameters $(C_k^{\mathtt{LO}}, \zeta_k, T_k)$ in an interval of length Q. Thus, the NC workload of sporadic task τ_k with parameters $(C_k^{\mathtt{LO}}, \zeta_k, T_k)$ within an interval of length Q can be given as $\mathsf{W}_k^{\mathtt{NC}}(Q)$ according to Eq. (9.2).

Within the interval $[r_k^y, r_i^x + t)$ in Figure 9.4, there are at most $(\mathcal{N}_{t,s}^k + 1)$ jobs of task τ_k that each executes for $C_k^{\mathtt{HI}}$ time units. Therefore, the workload of task τ_k within $[r_k^y, r_i^x + t)$ is equal to $(\mathcal{N}_{t,s}^k + 1) \cdot C_k^{\mathtt{HI}}$. The workload $\mathsf{P}_k(s,t)$ of task τ_k within the entire problem window $[r_i^x, r_i^x + t)$ for the reference pattern is given as follows:

$$\mathsf{P}_k(s,t) = \mathsf{W}_k^{\mathtt{NC}}(Q) \; + (\mathcal{N}_{t,s}^k + 1) \cdot C_k^{\mathtt{HI}} \tag{9.17}$$

In summary, the workload $\mathsf{P}_k(s,t)$ of task τ_k for the reference pattern is given using Eq. (9.16) and Eq. (9.17) for Case (A) and Case (B), respectively.

STEP 2 (net increase in workload due to shift): In this step, by shifting the problem window within the reference pattern in Figure 9.4 the ***maximum net increase*** in workload within the shifted problem window in comparison to $\mathsf{P}_k(s,t)$ is determined. According to the analysis by Bertogna and Cirinei in [BC07], the releases of the jobs in the reference pattern for Case (A) represents the worst-case workload of task τ_k with

parameters $(C_k^{\mathtt{HI}}, D_k, T_k)$ in any interval of length t. Therefore, it is only needed to consider shifting the problem window in the reference pattern for Case (B), i.e., whenever $Q > 0$. The *maximum net increase* in workload in addition to $\mathsf{P}_k(s,t)$ for all possible leftward and rightward shifts of the problem window in Figure 9.4 is given in Lemma 9.2 (proof is in Appendix A, page 225).

Lemma 9.2. *The net increase in workload due to any shift of the problem window in Figure 9.4 is bounded by* $(C_k^{HI} - C_k^{LO})$.

Given the workload $\mathsf{P}_k(s,t)$ for the reference pattern in Eq. (9.16) and Eq. (9.17) respectively for Case (A) and Case (B), the value of CI workload $\mathtt{WH}_k^{\mathtt{CI}}(s,t)$ of the HI-critical task τ_k in the problem window is given as follows:

$$\mathtt{WH}_k^{\mathtt{CI}}(s,t) \;=\; \begin{cases} \mathsf{P}_k(s,t) \;+ (C_k^{\mathtt{HI}} - C_k^{\mathtt{LO}}) & \text{if } \; Q > 0 \\ \mathsf{P}_k(s,t) & \text{otherwise} \end{cases} \tag{9.18}$$

In summary, the NC and CI workloads of a LO-critical task $\tau_k \in \mathtt{hpL(i)}$ within a problem window of length t of task τ_i are given in Eq. (9.8) and Eq. (9.9), respectively. And, the NC and CI workloads of a HI-critical task $\tau_k \in \mathtt{hpH(i)}$ within a problem window of length t of task τ_i are given in Eq. (9.11) and Eq. (9.18), respectively. Based on the NC and CI workloads of each task $\tau_k \in \mathsf{HP}_i \;= (\mathtt{hpL(i)} \cup \mathtt{hpH(i)})$, the response time $R_i^{\mathtt{HI}}$ of the HI-critical task τ_i is derived in next subsection.

9.5.3 The RTA Test for HI Criticality Level

The response time $R_i^{\mathtt{HI}}$ of HI-critical task τ_i is calculated by computing the interfering workload, total interfering workload and interference based on the workload of the tasks in HP_i within the problem window of task τ_i.

Interfering Workload. The NC and CI interfering load of task τ_k within the problem window of length t for some given s are denoted by $\overline{\mathtt{I}}_{k,i}^{\mathtt{NC}}(s,t)$ and $\overline{\mathtt{I}}_{k,i}^{\mathtt{CI}}(s,t)$ whenever τ_k is NC and CI task, respectively. An upper bound on the interfering workload of a higher priority task within the problem window is the workload of the higher priority task within that problem window. However, it is pointed out in [BC07, GSYY09, DB11b] that it is sufficient to consider the interfering workload of a higher priority task limited to at most $(t - C_i + 1)$ within the problem window size t whenever task τ_i has execution time C_i. The values of $\overline{\mathtt{I}}_{k,i}^{\mathtt{NC}}(s,t)$ and $\overline{\mathtt{I}}_{k,i}^{\mathtt{CI}}(s,t)$ are given as follows:

$$\overline{\mathtt{I}}_{k,i}^{\mathtt{NC}}(s,t) \;=\; \begin{cases} min\{\mathtt{WL}_k^{\mathtt{NC}}(s,t), t - C_i^{\mathtt{HI}} + 1\} & \text{if } \tau_k \in \mathtt{hpL(i)} \\ \\ min\{\mathtt{WH}_k^{\mathtt{NC}}(s,t), t - C_i^{\mathtt{HI}} + 1\} & \text{if } \tau_k \in \mathtt{hpH(i)} \end{cases}$$

$$\overline{\mathtt{I}}_{k,i}^{\mathtt{CI}}(s,t) \;=\; \begin{cases} min\{\mathtt{WL}_k^{\mathtt{CI}}(s,t), t - C_i^{\mathtt{HI}} + 1\} & \text{if } \tau_k \in \mathtt{hpL(i)} \\ \\ min\{\mathtt{WH}_k^{\mathtt{CI}}(s,t), t - C_i^{\mathtt{HI}} + 1\} & \text{if } \tau_k \in \mathtt{hpH(i)} \end{cases}$$

The difference between the CI and NC interfering workload of task τ_k is denoted by $\overline{\mathtt{I}}^{\mathtt{DIFF}}_{k,i}(s,t)$ and is given as:

$$\overline{\mathtt{I}}^{\mathtt{DIFF}}_{k,i}(s,t) = \overline{\mathtt{I}}^{\mathtt{CI}}_{k,i}(s,t) \; - \overline{\mathtt{I}}^{\mathtt{NC}}_{k,i}(s,t)$$

Total Interfering Workload. The upper bound on total interfering workload due to all the tasks in set $\mathtt{HP}_i$ within the problem window for some given s is denoted by $\overline{\mathtt{I}}_i(s,t)$. The value of $\overline{\mathtt{I}}_i(s,t)$ is given as follows:

$$\overline{\mathtt{I}}_i(s,t) = \sum_{\tau_k \in \mathtt{HP}_i} \overline{\mathtt{I}}^{\mathtt{NC}}_{k,i}(s,t) \; + \sum_{\tau_k \in Max(\mathtt{HP}_i \; , m-1)} \overline{\mathtt{I}}^{\mathtt{DIFF}}_{k,i}(s,t) \tag{9.19}$$

where $Max(\mathtt{HP}_i \; , m-1)$ is the set of $(m-1)$ tasks from set $\mathtt{HP}_i$ that have the largest values of $\overline{\mathtt{I}}^{\mathtt{DIFF}}_{k,i}(s,t)$.

Interference. Because interference is an integer and all the m processors are busy executing tasks from $\mathtt{HP}_i$ while task τ_i is interfered, the upper bound on interference due to the tasks in $\mathtt{HP}_i$ on any job of task τ_i within the problem window of length t is $\lfloor \overline{\mathtt{I}}_i(s,t)/m \rfloor$.

The Response Time Test. The response time of HI-critical task τ_i for some given s is given as follows:

$$R^{\mathtt{HI}}_{i,s} \leftarrow C^{\mathtt{HI}}_i + \lfloor \frac{\overline{\mathtt{I}}_i(s, R^{\mathtt{HI}}_{i,s})}{m} \rfloor \tag{9.20}$$

The Eq. (9.20) can be solved by iteratively searching the least fixed point starting with $R^{\mathtt{HI}}_{i,s} = C^{\mathtt{HI}}_i$ for the right-hand side of Eq. (9.20). The response time $R^{\mathtt{HI}}_i$ of task τ_i during any HI-criticality behavior of the system is given as:

$$R^{\mathtt{HI}}_i = \max_{0 \le s \le R^{\mathtt{LO}}_i} \{R^{\mathtt{HI}}_{i,s}\} \tag{9.21}$$

When certifying a system at HI criticality level, Eq. (9.21) can be used to determine whether the HI-critical task τ_i meets its deadline during all HI-criticality behaviors of the system. The RTA test in Eq. (9.21) is OPA-compatible since it does not depend on the relative priority ordering of the higher priority tasks in $\mathtt{HP}_i$ and all conditions given in page 83 for a schedulability test to be OPA-compatible are satisfied.

An Example: Consider the dual-criticality task set in Table 9.1 where task τ_1 is the *lowest* priority task. It is shown in subsection 9.4.1 that task τ_1 is schedulable for all LO-criticality behaviors and $R^{\mathtt{LO}}_1 = 2$. Since task τ_1 is a HI-critical task, i.e., $L_1 = \mathtt{HI}$, the aim is to calculate $R^{\mathtt{HI}}_1$ to verify if τ_1 is schedulable in all HI-criticality behaviors.

Calculating $R^{\mathtt{HI}}_1$: According to Eq. (9.21), the response time $R^{\mathtt{HI}}_1$ is the maximum of $R^{\mathtt{HI}}_{1,s}$ for all $s = 0, \ldots, R^{\mathtt{LO}}_1$ where $R^{\mathtt{LO}}_1 = 2$. The values of $R^{\mathtt{HI}}_{1,s}$ for all $s = 0, 1, 2$ are calculated using the recurrence in Eq. (9.20) in the table below.

The first column represents all possible values of s, $0 \leq s \leq R_1^{\mathsf{LO}}$. The second column presents the length of the problem window; initially, set to $R_{1,s}^{\mathsf{HI}} = C_1^{\mathsf{HI}} = 2$ for each new value of s in the first column. The third column presents (based on Eq. (9.19)), the total interfering workload $\overline{\mathtt{I}}_1(s, R_{1,s}^{\mathsf{HI}})$ of the higher-priority tasks τ_2 and τ_3 considering the length of the problem window given in the second column. Finally, the right hand side of Eq. (9.20), i.e., new value of $R_{1,s}^{\mathsf{HI}}$, is evaluated in the forth column.

s	$R_{1,s}^{\mathsf{HI}}$(window)	$\overline{\mathtt{I}}_1(s, R_{1,s}^{\mathsf{HI}})$	$R_{1,s}^{\mathsf{HI}} \leftarrow C_1^{\mathsf{HI}} + \lfloor \frac{\overline{\mathtt{I}}_1(s, R_{1,s}^{\mathsf{HI}})}{2} \rfloor$
0	$C_1^{\mathsf{HI}} = 2$	2	$2 + \lfloor \frac{2}{2} \rfloor = 3$
	3	2	$2 + \lfloor \frac{3}{2} \rfloor = 3$
1	$C_1^{\mathsf{HI}} = 2$	3	$2 + \lfloor \frac{3}{2} \rfloor = 3$
	3	3	$2 + \lfloor \frac{3}{2} \rfloor = 3$
2	$C_1^{\mathsf{HI}} = 2$	3	$2 + \lfloor \frac{3}{2} \rfloor = 3$
	3	3	$2 + \lfloor \frac{3}{2} \rfloor = 3$

It is evident that $R_{1,s=0}^{\mathsf{HI}} = 3$, $R_{1,s=1}^{\mathsf{HI}} = 3$, and $R_{1,s=2}^{\mathsf{HI}} = 3$ (see the shaded cells in the last column). Therefore, it follows that $R_1^{\mathsf{HI}} = 3$ based on Eq. (9.21). Since $R_1^{\mathsf{HI}} = 3 \leq D_1 = 3$, the deadline of task τ_1 is met in all the HI-criticality behaviors of the system. Therefore, task τ_1 meets all its deadlines in both LO and HI criticality behaviors of the system. And, the two other tasks τ_2 and τ_3 having higher priorities are trivially schedulable since $m = 2$. Consequently, the dual-criticality task set in Table 9.1 is MSM-schedulable. □

9.6 Schedulability Analysis for $\mathcal{L} > 2$

In this section, the main principle to compute the response time R_i^ℓ of task τ_i is presented considering the ℓ-criticality behavior of the system where $3 \leq \ell \leq L_i$ and $L_i \leq \mathcal{L}$.

Consider the problem window $[r_i^x, r_i^x + t)$ of some generic job J_i^x of task τ_i. Assume that $S = \{s_1, \ldots, s_{(\ell-1)}\}$ is the set of relative distances from r_i^x such that the system switches from ν-criticality to $(\nu+1)$-criticality behavior at time $(r_i^x + s_\nu)$ for each $s_\nu \in S$. For the sake of analysis, assume that $s_\nu = \infty$ for $\nu \geq \ell$, and $s_\nu = 0$ for $\nu = 0$.

According to the MSM algorithm, task $\tau_k \in \mathsf{HP}_i$ is allowed to execute within the problem window $[r_i^x, r_i^x + t)$ before time instant $(r_i^x + p)$ during the ℓ-criticality behavior of the system such that $p = min\{t, s_{L_k}\}$. In other words, if $L_k < \ell$, then task τ_k is allowed to execute before $(r_i^x + s_{L_k})$ in the problem window; otherwise, task τ_k is allowed to execute during the entire problem window for all ℓ-criticality behaviors of the system.

The response time of task τ_i for some *given* set S is denoted by $R_{i,S}^\ell$. The response time R_i^ℓ is the maximum $R_{i,S}^\ell$ over all possible sets S where each $s_\nu \in S$ can have

any value between $[0, R_i^v]$ and $s_\nu \leq s_{(\nu+1)}$. Therefore, the number of different sets S that one has to consider to find R_i^ℓ is upper bounded by $(D_i)^\mathcal{L}$. However, the number of different criticality levels $\mathcal{L}$ in many practical safety-critical systems is not very large, e.g., according to the RTCA DO-178B standard, there are five different Design Assurance Levels (DAL A to DAL E) for software in avionics systems, and according to ISO 26262 standard, the safety functions in automotive systems can have four different Automotive Safety Integrity Levels (ASIL A to ASIL D).

The response time $R_{i,S}^\ell$ can be derived (similar to that of in Section 9.5 for dual-criticality systems) once the NC and CI workloads of each task $\tau_k \in \mathsf{HP}_i$ in $[r_i^x, r_i^x + p)$ are known, where $p = min\{t, s_{L_k}\}$. The basic idea for calculating the NC and CI workloads of task $\tau_k \in \mathsf{HP}_i$ is presented next.

NC Workload: In order to find the NC workload of task τ_k within an interval of length p, consider that one job of task τ_k arrives exactly at the beginning of the window and subsequent jobs arrive and execute as early as possible. To find R_i^ℓ, the upper bound on NC workload of $\tau_k \in \mathsf{HP}_i$ within an interval of length p can be calculated as follows:

- If all the jobs of task τ_k executes for C_k^ℓ time units within an interval length p, then the total workload within the problem window is:

$$\mathsf{W}_k^{upper} = \lfloor p/T_k \rfloor \cdot C_k^\ell + min\{C_k^\ell,\ p - \lfloor p/T_k \rfloor \cdot T_k\}$$

- However, each of at least $\lfloor \frac{s_\nu}{T_k} \rfloor$ jobs of task τ_k executes for at most C_k^ν time units for $\nu = 1 \ldots (\ell - 1)$. This is because the system exhibits ν-criticality behavior before $(r_i^x + s_\nu)$. Thus, an upper bound on NC workload within the problem window is:

$$\mathsf{W}_k^{upper} - \sum_{\nu=1}^{\ell-1} \lfloor s_\nu / T_k \rfloor \cdot (C_k^{(\nu+1)} - C_k^\nu)$$

CI Workload: In order to calculate the CI workload within the problem window, consider the releases of the jobs of τ_k as follows (called, the *reference pattern*): one job of task τ_k releases exactly at $(r_i^x + p - C_k^\ell)$ and executes for C_k^ℓ time units as early as possible; and earlier jobs of τ_k are released and execute as *late* as possible.

Given the reference pattern, the release time of each job of task τ_k relative to the beginning of the interval $[r_i^x, r_i^x + p)$ can be determined. For each such job of task τ_k, say job J_k^y, that executes within the problem window, the *largest* s_ν, if one exists in S, such that $r_k^y \leq (r_i^x + s_\nu) < r_k^{(y+1)}$, can be determined. If such an $s_\nu \in S$ exists for job J_k^y, then it is assumed that job J_k^y executes for $C_k^{(\nu+1)}$ time units. If no such s_ν exists for job J_k^y, then it is considered that job J_k^y executes for $C_k^{(\overline{\nu}+1)}$ time units where $(r_i^x + s_{\overline{\nu}})$ is the *closest* criticality-switch time of the system prior to the release of job J_k^y (such an $s_{\overline{\nu}}$ must exist since it is assumed that $s_0 = 0$).

Given the execution time of each job of task τ_k for the reference pattern within the interval of length p, the workload of task τ_k for the reference pattern can be computed.

And, it can be shown that the maximum increase in workload due to any possible shift of the problem window within the reference pattern is bounded by $(C_k^\ell - C_k^1)$. By adding these two workload factors, the CI workload within the problem window is derived. Once the CI and NC workloads of each task $\tau_k \in \mathsf{HP}_i$ are known, the recurrence for the response time $R_{i,S}^\ell$ can be derived by finding the interfering workload, total interfering workload and interference for a given set S.

9.6.1 Finding Priorities using Audsley's Algorithm

The pseudocode for applying Audsley's approach to find the fixed-priority ordering of the MC tasks is given in Figure 9.5. The MC tasks in set Γ are assigned priority starting from the lowest priority level n to the highest priority level 1 using the outer loop in line 1. If $R_i^\ell \leq D_i$ for all $\ell \leq L_i$ for some priority-unassigned task τ_i (i.e., condition in line 3–5 is true), then task τ_i is assigned the current priority level in line 6. The value of R_i^ℓ is calculated in line 3–5 by assuming priority level PL for the priority-unassigned task τ_i and higher priorities for all other priority-unassigned tasks.

If some priority-unassigned task τ_i is assigned the current priority level in line 6, then the priority assignment for next (higher) priority level is considered (i.e., next iteration of the outer loop starts). If no priority-unassigned task can be assigned the current priority level (condition in line 3 is false for all priority-unassigned tasks), then the priority assignment fails and line 8 reports "Failure". If all the tasks are assigned priorities, then line 9 reports "Success".

Algorithm OPA(Mixed-Criticality task set Γ)

1. for each priority level PL, lowest first
2. for each priority-unassigned task $\tau_i \in \Gamma$
3. If $R_i^\ell \leq D_i$ for all $\ell \leq L_i$, where task τ_i is assumed to have
4. priority level PL with all other priority-unassigned
5. tasks are assumed to have higher priorities, Then
6. assign τ_i priority level PL
7. break (continue outer loop)
8. return "Failure"
9. return "Success"

Figure 9.5: *OPA algorithm for MC tasks scheduled using MSM.*

Time-Complexity for Dual-Criticality. To determine whether a dual-criticality task τ_i meets all the deadlines in all correct behaviors of the system, it is required to find R_i^{LO} and R_i^{HI} based on Eq. (9.7) and Eq. (9.21), respectively. Since the recurrence in Eq. (9.7) can be solved in $O(T_{max})$ iterations, the time complexity to find R_i^{LO} is $O(T_{max})$, where T_{max} is the largest period of the task set.

To compute R_i^{HI} based on Eq. (9.21), one has to evaluate the recurrence in Eq. (9.20) for each possible value of s, where $0 \leq s \leq R_i^{\mathsf{LO}}$. The recurrence in Eq. (9.20) can be

solved for a *given* value of s using $O(T_{max})$ iterations. Since $s \leq R_i^{\mathsf{LO}} \leq T_{max}$, at most $O(T_{max}^2)$ iterations are needed to find R_i^{HI} based on Eq. (9.21). Therefore, the time complexity to find R_i^{LO} and R_i^{HI} is pseudo-polynomial.

When applying the OPA algorithm in Figure 9.5 for dual-criticality system, evaluating the condition in line 3–5 requires to compute R_i^{LO} and R_i^{HI} for at most n different tasks at priority level $\mathsf{PL} = n$, for at most $(n-1)$ different tasks at priority level $\mathsf{PL} = (n-1)$, and so on. Therefore, the total number of times line 3–5 is executed is $O(n^2)$. Therefore, the time complexity of the OPA algorithm for dual-criticality system is $O(n^2 \cdot T_{max}^2)$ which is pseudo-polynomial in the representation of the task set. It can be shown that the time complexity of the OPA algorithm for task set with $\mathcal{L}$ criticality levels is $O(n^2 \cdot \mathcal{L} \cdot T_{max}^{\mathcal{L}})$ which can be considered pseudo-polynomial for any fixed value of $\mathcal{L}$ that is reasonable for practical mixed-criticality systems.

9.7 Empirical Investigation

In this section, the result of empirical investigation to measure the performance of the proposed response time test for dual-criticality systems is presented. In particular, the effectiveness of the OPA-based priority assignment scheme (as given in Figure 9.5) is compared with the following two heuristic priority assignment schemes:

- Deadline-Monotonic Priority Ordering (`DMPO`): The priorities are ordered based on deadline (i.e., the shorter the relative deadline, the higher is the priority).
- Criticality-Monotonic Priority Ordering (`CMPO`): The priorities are first ordered based on criticality (i.e., `HI` critical task first); and then based on deadline (i.e., shorter relative deadline first).

To determine the `MSM`-schedulability of randomly generated task sets using OPA, `DMPO`, and `CMPO` priority assignment schemes, the response-time tests in Eq. (9.7) and Eq. (9.21) are used. The well-known metric, called *acceptance ratio*, is used to evaluate the effectiveness of different priority assignment schemes. The acceptance ratio of a priority assignment scheme is the percentage of the randomly generated task sets that are deemed schedulable using the response-time tests in Eq. (9.7) and Eq. (9.21) at a given utilization level. Before presenting the experimental results, the task set generation algorithm is presented next.

Task set Generation. The `UUnifast-Discard` algorithm proposed by Davis and Burns [DB11b] (given in subsection 5.6.1, page 66) is used to generate utilizations for n tasks with total utilization equal to U. Once a set of n utilizations $\{u_1, u_2, \ldots u_n\}$ of a task set is generated, the other parameters of each task τ_i are generated as follows:

- The minimum inter-arrival time T_i of each task τ_i is generated from the uniform random distribution within the range $[1ms, 1000ms]$.
- The `LO`-criticality execution time of task τ_i is set to $C_i^{\mathsf{LO}} = u_i \cdot T_i$. Note that u_i is the utilization corresponds to the task's `LO`-criticality execution time.

- Whether a generated task is a LO- or HI-critical task is determined using a simulation parameter CP where CP ≤ 1. A random number in the range $[0, 1]$ is generated. If this newly generated random number is greater than CP, then task τ_i is a LO-critical task; otherwise, the task is HI-critical.

- The HI-criticality execution time of τ_i is set to $C_i^{\mathsf{HI}} = C_i^{\mathsf{LO}} \cdot \mathsf{CF}$, where CF is a simulation parameter ≥ 1.

- The relative deadline D_i of task τ_i is generated from the uniform random distribution within the range $[C_i^{\mathsf{LO}}, T_i]$ and $[C_i^{\mathsf{HI}}, T_i]$ whenever τ_i is a LO-critical and HI-critical task, respectively.

Each of the experiments is characterized by a 4-tuple $(m, n, \mathsf{CF}, \mathsf{CP})$ where m is the number of processors, n is the task set size, CF is equal to $\frac{C_i^{\mathsf{HI}}}{C_i^{\mathsf{LO}}}$, and CP corresponds to the percentage of HI-critical tasks in a task set. For each experiment, total 40 different utilization levels $\{0.025m, \ldots 0.975m, m\}$ are considered. For each utilization level $U \in \{0.025m, \ldots 0.975m, m\}$, total 1000 task sets are generated with parameters n, CF, CP and U.

Result Analysis. Experiments with different simulation parameters $m \in \{2, 4, 8\}$, $n \in \{10, 20, 40, 60\}$, $\mathsf{CF} = \{2, 3, 4\}$ and $\mathsf{CP} = \{0.25, 0.5, 0.75\}$ for both implicit-deadline and constrained-deadline task sets are conducted.

The acceptance ratios for experiment ($m = 4, n = 20, \mathsf{CF} = 2, \mathsf{CP} = 0.5$) considering the OPA, DMPO, and CMPO priority-assignment schemes are presented in Figure 9.6 (the trend is similar for other experiments). The x-axis represents the system utilization (i.e., U/m) and the y-axis represents the acceptance ratios.

Since the scheduling window for implicit-deadline task sets is relatively wider than that of the constrained-deadline task sets, the acceptance ratios of all priority assignment schemes for implicit-deadline task sets are relatively better in comparison to the constrained-deadline task sets in Figure 9.6. The performance of CMPO is very poor in comparison to the DMPO scheme, i.e., the criticality-monotonic priority ordering is far from the optimal priority assignment scheme. The acceptance ratio of the OPA scheme is more than 50% larger than that of the DMPO scheme at $0.6m$ and $0.4m$ utilization levels for implicit-deadline and constrained-deadline task systems, respectively. The OPA scheme significantly outperforms both the DMPO and CMPO schemes.

It is not difficult to realize that the acceptance ratio would be relatively lower for experiments with relatively larger CF and/or CP. This is because larger CF and/or CP means larger total utilization of the HI-critical tasks; and it is generally difficult to schedule task sets having large total utilization.

The acceptance ratios of the OPA scheme using $\mathsf{CF} = 2$ and $\mathsf{CP} = 0.5$ for implicit-deadline task sets for different (m, n) pairs are presented in Figure 9.7. There are variations in acceptance ratios at higher U for the variations in m and n. The reasons for such variations are also common for traditional global FP scheduling and discussed in Chapter 6. However, the acceptance ratios for all the cases in Figure 9.7 are 100% upto

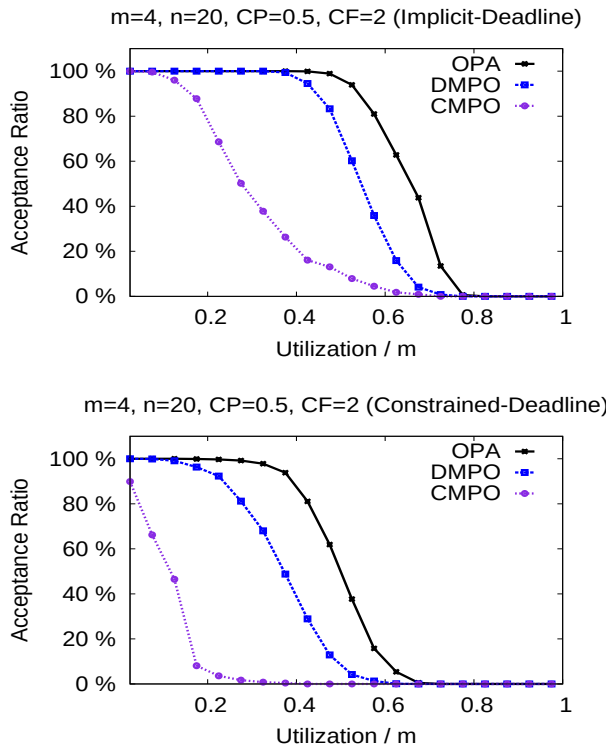

Figure 9.6: *Acceptance ratios for $D_i = T_i$ (top) and $D_i \leq T_i$ (right)*

$0.4m$ utilization level which justifies the scaleability of the proposed response time test combined with the OPA algorithm.

9.8 Related Works

The seminal work by Vestal in [Ves07] first proposed the MC task model and its analysis based on FP scheduling algorithm on uniprocessor platform. Vestal's algorithm is proved as the optimal for traditional FP scheduling on uniprocessor by Dorin et al. [DRRG10]. By showing that neither FP nor EDF scheduling of MC tasks on uniprocessor dominates the other, Baruah and Vestal proposed a hybrid algorithm by combining the benefits of both FP and EDF policies [BV08]. Recently, a variant of FP scheduling algorithm and its analysis on uniprocessor platform is proposed by Baruah et al. based on the following observation [BBD11b]: the run-time monitoring of execution time of the jobs can be used to drop jobs of ℓ-critical tasks as soon as the system switches to $(\ell+1)$-criticality behavior. The MSM algorithm proposed in this chapter also uses this observation but for multiprocessors.

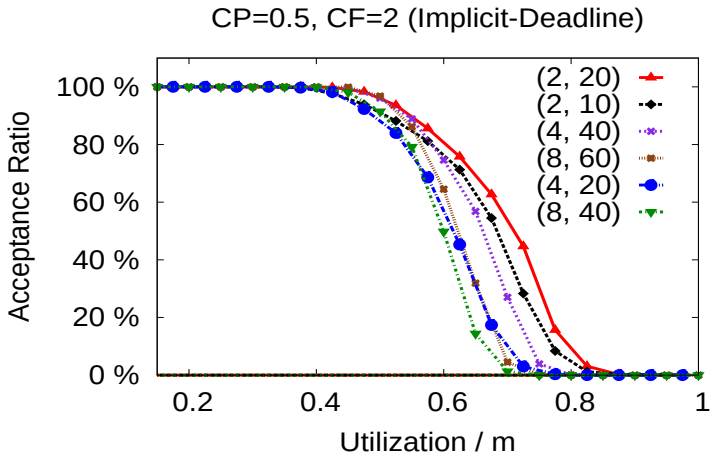

Figure 9.7: *Acceptance ratio using OPA scheme with different* (m, n) *pairs.*

Several works addressed MC scheduling of a finite collection of jobs on uniprocessors. It has been proved by Baruah et al. [BBD+12b] that determining the feasibility of a collection of MC jobs is strongly NP-hard, even when all release times are identical and there are only two criticality levels. Baruah et al. proposed Own Criticality Based Priority (OCBP) algorithm for scheduling a finite collection of jobs on uniprocessor. Algorithm OCBP works as follows: jobs are assigned fixed-priorities in offline, and the highest priority ready job is always dispatched at run-time [BLS10]. The processor speed-up factor of the OCBP algorithm for dual-criticality system is 1.619, i.e., any feasible instance of dual-criticality jobs on unit-capacity processor is also OCBP-schedulable on a processor that is 1.619 times faster [BLS10]. An improved load-based sufficient schedulability condition of the OCBP algorithm is proposed in [LB10b] by Li and Baruah.

By assuming the earliest releases of the jobs within a *busy interval*, Li and Baruah proposed interesting techniques to apply the OCBP algorithm for scheduling sporadic MC tasks on uniprocessor platform [LB10a]. However, Due to the sporadic nature of the tasks, the priorities of the jobs are recomputed at run-time and such priority recomputation at run-time has pseudo-polynomial time complexity [LB10a]. Recently, Guan et al. [GESY11b] proposed a novel polynomial time algorithm for recomputing the priorities at run-time for scheduling sporadic tasks using the OCBP algorithm.

An EDF based scheduling algorithm, called EDF-VD (Virtual-Deadline), in which the deadlines of the implicit-deadline sporadic tasks are modified online, is proposed by Baruah et al. in [BBD+11a]. The algorithm EDF-VD modifies the deadlines of the tasks depending of the behavior of the system at different criticality levels and schedule the tasks based on EDF scheduling according to the modified deadlines. The processor speed-up factor of EDF-VD scheduling for dual-criticality system is 1.619. By performing a more precise analysis of the EDF-VD scheduling of implicit-deadline MC sporadic tasks, the speed-up factor of EDF-VD is further improved by Baruah et al. to

1.333 [BBD+12a]. Ekberg and Yi [EY12] recently proposed interesting technique to compute the demand-bound [BMR90] function to determine the EDF schedulability of constrained-deadline MC sporadic tasks. The demand-bound of the tasks at each criticality level is determined by adjusting the deadline of the tasks when the system switches from LO to HI criticality behavior. The purpose of shaping or adjusting the demand is to respect the supply-bound [MFC01] function of the underlying uniprocessor platform to ensure schedulability.

Time-triggered (TT) scheduling of MC jobs on uniprocessor platform is proposed by Baruah and Fohler in [BF11]. The TT-scheduling essentially computes in offline, for each criticality levels, the scheduling table that stores the time instant at which jobs will be dispatched for execution. When the criticality behavior of the system switches from ℓ to $(\ell+1)$, then jobs are scheduled based on the scheduling table computed for criticality level $(\ell+1)$. The processor speed-up factor for TT-scheduling is 1.619.

Many of the scheduling algorithms for MC systems considers dropping tasks of lower criticality levels when the system switches to a higher criticality level. However, the lower criticality tasks may not need to be dropped as long as they are not causing a higher criticality task to miss its deadline. Based on this observation, Santy et al. [SGTG12] proposed a method, called Latest Completion Time (LCT), that allows lower criticality task to execute using uniprocessor FP scheduling until time instant at which the lower criticality task is suspended to allow execution of a higher criticality task to avoid missing its deadline. The lower-criticality task may resume its execution later when the system switches back to lower-criticality behavior.

The only work that considers multiprocessor scheduling of MC system is proposed by Li and Baruah in [LB12] but for implicit-deadline tasks. This work is based on the basic principle of computing the deadlines for uniprocessor EDF-VD scheduling but uses the utilization-bound test of global dynamic-priority scheduling, known as fpEDF, proposed by Baruah in [Bar04]. The processor speed-up factor for this algorithm is $(\sqrt{5}+1)$: a MC task sets that can be scheduled in a certifiably correct manner on m unit capacity processors by an optimal clairvoyant scheduling algorithm can be scheduled by the proposed algorithm on m speed-$(\sqrt{5}+1)$ processors. This work in [LB12] considers implicit-deadline tasks, dynamic priority and is applicable to only two criticality levels. The work presented in this chapter is the first work that considers global FP scheduling of certifiable mixed-criticality sporadic tasks with constrained deadlines and more than two criticality levels on multiprocessor platform.

Many other works addressed scheduling of MC systems for aspects other than certification. Pellizzoni et al. [PMN+09] and Petters et. al. [PLHE09] proposed techniques for isolating (either in time or space) subsystems having different criticality levels based on reservation based approach. However, these work concentrate on providing isolation through worst-case reservation of resources and do not efficiently utilize the resources. The work proposed by De Niz et al. [dNLR09] observed that isolation among multiple subsystems that are based on reservation based approach may suffer from, so called *criticality inversion* problem: the deadline of a higher-criticality job may be missed while allowing a lower criticality job to meet its deadline. In addition, as-

signing priorities based on criticality to avoid criticality-inversion is not a good priority assignment policy for meeting the deadlines. They have proposed *slack-aware* scheduling that dynamically assigns the priorities to tasks or jobs to avoid criticality inversion while focusing on efficient use of the resources [dNLR09]. This algorithm avoids criticality inversion under which low-criticality task can not interfere with high-criticality task but high-criticality task can steal cycles from the low-criticality task under overload situations to meet deadlines. The work in [dNLR09] is further extended for non-preemptable shared resources [LdNRM10] and distributed systems [LdNR11]. Mollison et al. [MEA+10] proposed an architecture for scheduling MC tasks based on criticality-monotonic scheduling on multicore. The allocation of MC tasks in a distributed systems is considered in [TSP11], where each task allocated to a processor is given a time partition by determining the sequence and size of each partition in addition to finding the scheduling table for each processor.

9.9 Summary

In this chapter, the global FP scheduling of mixed-criticality systems on preemptive multiprocessors is considered. In order to utilize the processors efficiently and to facilitate certification, a sufficient schedulability test based on response-time analysis of the proposed MSM algorithm is derived. This schedulability test can also be used to find fixed-priority ordering of the MC tasks based on Audsley's approach. The time-complexity for evaluating the proposed test is pseudo-polynomial for dual-criticality system. In addition, the proposed test is applicable to system having more than two criticality levels which makes the algorithm relevant for many practical safety-critical systems that have more than two criticality levels. The schedulability test of the MSM algorithm can be easily extendented by finding a better priority assigning policy using the separation criteria proposed in Chapter 6 and using the HPA-based priority assignment policy.

In order to design a certification-cognizant scheduling algorithm for mixed-criticality systems, the criticality behaviors of the systems need to be monitored at run-time. However, such monitoring requires to know what behavior specifies a particular criticality-behavior of the system. The criticality-behavior of the system is determined based on the run-time behavior of the system which varies from one time instant to another. The run-time behavior of the system depends on many factors, for example, the actual execution time of each task, the actual inter-arrival time of each task, energy consumption, the frequency and types of faults, and so on. This chapter considers one such source of variation to determine the criticality-behavior at runtime: the actual execution time of each task. By appropriately modeling the criticality-behavior based on other sources of variations that specify the criticality behavior, designing new certification-cognizant real-time scheduling algorithms for MC systems is left as future work.

10 Conclusion

This thesis deals with the modeling, analysis, and verification of three important non-functional behaviors of real-time systems: timeliness, fault tolerance, and mixed criticality. The level of acceptability or the desired quality of each non-functional behavior is modeled as a set of design constraints — satisfaction of which are important for correctness, popularity, and competitiveness of the system. The functional behaviors (i.e., the workload) of the real-time applications are modeled using constrained-deadline sporadic tasks that are dispatched for execution on a platform having multiple identical processors/cores using global fixed-priority scheduling algorithm. The non-functional behaviors considered in this thesis are common in many safety-critical real-time systems; therefore, the proposed scheduling algorithms and the corresponding schedulability tests have wide applicability for many practical systems.

The acceptability of timeliness behavior is modeled as hard deadline for each sporadic task. The proposed schedulability tests for global FP scheduling verify offline whether all the deadlines of all the tasks are met or not. The acceptability of fault-tolerant behavior is modeled based on the number and types of faults that need to be tolerated during the execution of the tasks. The proposed fault-tolerant scheduling algorithms have the responsibility to ensure that the effects of faults are mitigated in order to generate the correct output before the deadline of each task. Finally, the acceptability of mixed-criticality behavior is modeled as the level of assurance needed in meeting the deadlines of the tasks where different WCETs (estimated at different level of assurance) for each task are considered. The reason for considering certain level of assurance in meeting the deadlines of the tasks is to facilitate certification while efficiently utilizing the processing platform of the mixed-criticality system. To this end, the following three research questions are addressed in this thesis:

Q1 *(Timeliness) How to guarantee that all the deadlines of a real-time application are met on a particular computing platform?*

Q2 *(Fault Tolerance) How to guarantee that all the deadlines of a real-time application are met on a particular computing platform while providing fault-tolerance?*

Q3 *(Mixed Criticality) How to guarantee that all the deadlines of a real-time application are met while ensuring certification of mixed-criticality system at each criticality level?*

In this thesis, timeliness is about meeting the deadlines of the tasks; fault-tolerance is about providing correct service even in the presence of faults while also meeting the deadlines; and mixed-criticality is about certification (i.e., guaranteeing timeliness) regarding the integration of multi-criticality tasks on a common computing platform where different WCETs of each task are considered at varying degrees of confidence.

The purpose of modeling the real-time application and its design constraints is to ensure through analysis and verification that the system is predictable at runtime. A system is considered to be predictable when all the design constraints are satisfied for the assumed model of the system. Satisfying the temporal constraints (i.e., meeting the deadlines) is the main design constraint considered in this thesis. The temporal constraints of meeting the deadlines might be contending with the design constraints of other non-functional behaviors (e.g., fault-tolerance, criticality). In order to verify offline that whether all the design constraints will be met or not, schedulability tests are proposed by analyzing global FP scheduling. The proposed schedulability tests do not only dominate but also empirically perform significantly better than the corresponding state-of-the-art schedulability tests.

The different techniques used to analyze one particular non-functional behavior are orthogonal to the analysis of other non-functional behaviors in this thesis. For example, the criteria to determine the set of tasks to be kept separated from the schedulability analysis of a lower priority task (as proposed for the `IA-DA` test) can also be used for the schedulability analysis of the `FTGS` and `MSM` algorithms. Similarly, if a mixed-criticality system is also a fault-tolerant system, then the response-time based schedulability test of the `MSM` algorithm can be extended with the schedulability analysis used for the fault-tolerant `FTGS` algorithm in order to derive a new schedulability test.

The mathematical expressions of the proposed schedulability tests incorporate the parameters of the task set, processing platform, and design constraints. The compact representation and the efficiency in evaluating the proposed schedulability tests enable the designers making the trade-off between resource-requirement and rigidity of the design constraints. The analysis of the scheduling algorithms aims to reduce the pessimism in order to derive more effective schedulability tests for global FP scheduling. Such reduced pessimism is beneficial in reducing resource consumption and enables quick adaptation to changes, for example, adding new services on existing hardware.

Although the proposed algorithms consider fixed-priority scheduling of constrained-deadline tasks on multiprocessors, the corresponding results can be extended for other

work-conserving scheduling algorithms, for example, global EDF scheduling. To perform the schedulability analysis of global EDF scheduling, the technique for workload computation of the higher priority jobs within the problem window of each task has to be derived. In global EDF, each job having its absolute deadline in a problem window, that ends at the deadline of the analyzed task, becomes a contributor to the workload in that problem window. Depending on the non-functional behavior under study, the workload within the problem window has to be appropriately calculated. By finding the workload of the higher priority jobs, an upper bound on the interference on each task within its problem window can be calculated and a schedulability test for global EDF can be derived. In addition, designing new scheduling algorithms, performing precise schedulability analysis and deriving efficient schedulability tests for the following open problems are left as future work:

- There is an important source of pessimism in the existing schedulability analysis of global scheduling algorithms, which is stated as follows: **when a lower priority task τ executes, all the other $(m-1)$ processors are assumed to be idle**. This assumption is not always true as will be demonstrated now using an example.

  Consider the global FP scheduling of four tasks $\{\tau_1, \tau_2, \tau_3, \tau_4\}$ on $m = 2$ processors, where a task with lower index has higher priority. Also consider that the interference on task τ_3 according to the DA-LC test within a problem window of length D_3 is $(D_3 - C_3)$. Evidently, task τ_3 is guaranteed to be schedulable according to the DA-LC test. Now assume that $D_4 = D_3$, $C_3 = 4$, and $C_4 \leq C_3$. The total interfering workload within a problem window of length D_4 is at least $[m \cdot (D_3 - C_3) + 4]$ when analyzing the schedulability of task τ_4 based on the DA-LC test. By assuming that all the other processors are idle when task τ_4 executes within a problem window of length D_4, the interference on task τ_4 according to the DA-LC test is at least $(D_3 - C_3 + 2) = (D_4 - C_4 + 1)$. Therefore, the schedulability of task τ_4 can not be guaranteed based on the DA-LC test. However, the DA-LC test assumes that $(m-1) = 1$ processor is idle when task τ_3 executes, and therefore, task τ_4 is also schedulable since its relative deadline is equal to D_3 and its execution time is smaller than the execution time of task τ_3.

  The lesson learned is that the assumption that $(m-1)$ processors are idle, when a particular task executes, does not need to be enforced during the schedulability analysis of each task. Relaxing this assumption for appropriate tasks will result in more precise schedulability analysis and better schedulability test. Finding the details when such assumption can be relaxed is left as a future work.

- The fault-tolerance scheduling algorithms proposed in this thesis considers a fault model in which a particular job of each task is assumed to be affected by at most f task errors. A relatively general fault model would be to consider different number of task errors to be tolerated for different tasks. This is a more reasonable fault model since not every task is equally prone to the same number of errors. For example, a piece of complex software is possibly more prone to design errors than a simple software. In addition, the internal robustness in masking faults or er-

rors of different software can be different due to the difference in software design process, testing, debugging, and so on. Therefore, it is more reasonable to consider different number of errors to be tolerated for different tasks. Fault-tolerant schedulability analysis considering such a relatively general fault-model and relaxing the assumption of no-fault-propagation is left as a future work. In addition, schedulability analysis on multiprocessors considering checkpoint or imprecise-computation for error recovery is also another interesting future work.

- In order to provide different degrees of assurance needed in meeting the deadlines of mixed-criticality tasks, this thesis considers only one source of variation in the run-time behavior of the system, i.e., the execution time of each task. There are other sources of variations that may impact the degree of assurance needed for certifying a mixed-criticality system at various criticality levels. One such source is the inter-arrival time (period) of each task.

  The system designer may assume a relatively larger period of a task while the CA being more pessimistic may assume a shorter period of the same task. For example, consider an aircraft that periodically runs some diagnostic function to check if lightning (or some other disturbance) has caused some damages to the on-board electrical and electronic systems. The system designer may decide to execute the function in every minute whereas the CA may require to execute it every 5 seconds. To put it in another way, consider that the function is executed every minute during sunny weather and every 5 seconds during cloudy weather. Developing scheduling algorithm and schedulability test considering different periods along with different WCETs of each task at different criticality levels is another interesting future work. Similarly, the number and types of faults that may need to be tolerated for each task can be different for different criticality levels. Fault tolerant scheduling of `MC` systems considering different number and types of faults to be tolerated at different criticality levels is another interesting future work.

The research presented in this thesis is to help the system designers to build a predictable system. To this end, I wonder whether it is really possible to design a computerized system that is completely predictable. The answer is positive if the model of the system is perfect and the analysis of the system based on this "perfect" model is precise. Then, the question arises is whether the model of a computer system is perfect in capturing the environment of the system. I believe that it is really difficult to entirely capture the environment of computerized systems which may consist of:

- hardware (e.g., sensor, actuator, processing platform, accelerators, GPUs),
- software (e.g., application tasks, operating systems, middleware, drivers),
- inputs (e.g., from sensors, human users, other systems),
- users' interactions (e.g., robots, human beings),
- factors related to atmosphere (e.g., radiation, temperature, lightning, dust, snow),
- factors related to software design (e.g., competence, experience, testing).

In addition, changes in technology (e.g., introduction of multicore, miniaturization of transistors), changes in users' perceived level of comfort (e.g., autonomous cars), new operating condition/atmosphere (e.g., spacecraft in a new planet), and new certification standards — all are contributing to the difficulty in the design of predictable computerized systems. Perfect modeling and precise analysis considering all these sources of variability are daunting tasks in terms of time and complexity. Although it seems that we are far from building true predictable system, there are computerized systems that are in fact behaving predictably.

A computer system can hardly be entirely predictable and there are only systems which may have not yet become unpredictable and we can only design a "more" predictable system in comparison to another existing system. One way to build a *more* predictable system is to consider the different system layers — starting from the application to the middleware, operating system, processors and all the way down to the transistors — as *information providers* rather than *information concealers*. An interdependent system design approach in which information from one design layer is heavily exploited in another can help building a better predictable real-time system.

Bibliography

[AB98] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proc. of the RTSS*, pages 4 –13, 1998.

[AB08] B. Andersson and K. Bletsas. Sporadic Multiprocessor Scheduling with Few Preemptions. In *Proc. of the ECRTS*, pages 243–252, 2008.

[ABB96] N.C. Audsley, I.J. Bate, and A. Burns. Putting fixed priority scheduling theory into engineering practice for safety critical applications. In *Proc. of the RTAS*, pages 2 –10, 1996.

[ABB08] B. Andersson, K. Bletsas, and S. Baruah. Scheduling Arbitrary-Deadline Sporadic Task Systems on Multiprocessors. In *Proc. of the RTSS*, pages 385–394, 2008.

[ABJ01] B. Andersson, S. Baruah, and J. Jonsson. Static-Priority Scheduling on Multiprocessors. In *Proc. of RTSS*, pages 193–202, 2001.

[ABR+93] 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.

[ABRW91] N.C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline-Monotonic Approach. In *Proc. IEEE Workshop on Real-Time Operating Systems and Software*, pages 133–137, 1991.

[AFK05] J. Aidemark, P. Folkesson, and J. Karlsson. A Framework for Node-Level Fault Tolerance in Distributed Real-Time Systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 656–665, 2005.

[AJ] B. Andersson and J. Jonsson. Some insights on fixed-priority pre-emptive non-partitioned multiprocessor scheduling. In *In Proc. RTSS Work-in-Progress Session, Nov. 2000*.

[AJ03] B. Andersson and J. Jonsson. The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%. In *Proc. of ECRTS*, pages 33–40, 2003.

[ALRL04] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. on Depend. and Sec. Comp.*, 1(1):11–33, 2004.

[And08a] Björn Andersson. Global Static-Priority Preemptive Multiprocessor Scheduling with Utilization Bound 38%. In *Proc.of OPODIS*, pages 73–88, 2008.

[And08b] Björn Andersson. The Utilization Bound of Uniprocessor Preemptive Slack-Monotonic Scheduling is 50%. In *Proc. of ACM Symp. On Applied Computing*, pages 281–283, 2008.

[And10] B. Andersson. Conjecture about global fixed-priority preemptive multiprocessor scheduling of implicit-deadline sporadic tasks: The utilization bound of $SM - US(\sqrt{2} - 1)$ is $\sqrt{2} - 1$. In *In Proceedings of the 1st International Real-Time Scheduling Open Problems Seminar, in conjunction with the ECRTS*, 2010.

[AOMS00] R. Al-Omari, G. Manimaran, and Arun K. Somani. An efficient backup-overloading for fault-tolerant scheduling of real-time tasks. In *Proc. of the Workshops on Parallel and Distributed Processing*, pages 1291–1295, 2000.

[AOSM01] R. Al-Omari, Arun K. Somani, and G. Manimaran. A New Fault-Tolerant Technique for Improving Schedulability in Multiprocessor Real-time Systems. In *Proc.of the IPDPS*, page 8, 2001.

[ARI] ARINC Incorporated. ARINC specification 651: Design guidance for integrated modular avionics, November 1997.

[AS04] James H. Anderson and Anand Srinivasan. Mixed pfair/erfair scheduling of asynchronous periodic tasks. *J. Comput. Syst. Sci.*, 68(1):157–204, 2004.

[AS06] Armando Aguilar-Soto. *Fixed-Priority Scheduling Algorithms with Multiple Objectives in Hard Real-Time Systems*. PhD thesis, Department of Computer Science, The University of York, UK, 2006.

[AT06] B. Andersson and E. Tovar. Multiprocessor Scheduling with Few Preemptions. In *Proc. of the RTCSA*, pages 322–334, 2006.

[Aud91] N.C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. *Technical Report YCS 164, Dept of Computer Science, University of York, UK*, 1991.

[Aud01] N. C. Audsley. On Priority Assignment in Fixed Priority Scheduling. *Info. Proc. Letters*, 79(1):39–44, 2001.

[AUT] AUTOSAR, Automotive Open System Architecture, www.autosar.org.

[Avi85] A. Avižienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, 1985.

[Ayd07] H. Aydin. Exact Fault-Sensitive Feasibility Analysis of Real-Time Tasks. *IEEE Trans. on Comp.*, 56(10):1372–1386, 2007.

[BA10] Björn B. Brandenburg and James H. Anderson. Spin-based reader-writer synchronization for multiprocessor real-time systems. *Real-Time Syst.*, 46(1):25–87, September 2010.

[Bak06] T. P. Baker. An Analysis of Fixed-Priority Schedulability on a Multiprocessor. *Real-Time Systems*, 32(1-2):49–71, 2006.

[Bar04] S.K. Baruah. Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors. *IEEE Transactions on Computers*, 53(6):781 – 784, june 2004.

[Bar07] Sanjoy Baruah. Techniques for multiprocessor global schedulability analysis. In *Proc of RTSS*, pages 119–128, 2007.

[Bau05] R. Baumann. Soft errors in advanced computer systems. *IEEE Design and Test of Computers*, 22(3):258–266, 2005.

[BB05] Enrico Bini and Giorgio C. Buttazzo. Measuring the Performance of Schedulability Tests. *Real-Time Systems*, 30:129–154, 2005.

[BBA10] Andrea Bastoni, Bjorn B. Brandenburg, and James H. Anderson. An Empirical Comparison of Global, Partitioned, and Clustered Multiprocessor EDF Schedulers. In *Proc. of RTSS*, pages 14–24, 2010.

[BBA11] Andrea Bastoni, Bjorn B. Brandenburg, and James H. Anderson. Is semi-partitioned scheduling practical? In *Proc. of ECRTS*, pages 125–135, 2011.

[BBB+] J. Barhorst, T. Belote, P. Binns, J. Hoffman, J. Paunicka, P. Sarathy, J. S. P. Stanfill, D. Stuart, and R Urzi. White paper: A research agenda for mixed-criticality systems, april 2009, available at `http://www.cse.wustl.edu/~cdgill/CPSWEEK0_MCAR`.

[BBD+11a] Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Alberto Marchetti-Spaccamela, Suzanne Van Der Ster, and Leen Stougie. Mixed-criticality Scheduling of Sporadic task Systems. In *Proc. of the European conference on Algorithms*, pages 555–566, 2011.

[BBD11b] Sanjoy Baruah, Alan Burns, and Robert Davis. Response-time analysis for mixed criticality systems. In *Proc. of RTSS*, pages 34–43, 2011.

[BBD+12a] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *Proc of ECRTS*, pages 145 –154, 2012.

[BBD+12b] S. Baruah, V. Bonifaci, G. D'Angelo, Haohan Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling Real-Time Mixed-Criticality Jobs. *IEEE Transactions on Computers*, 61(8):1140 –1152, 2012.

[BC07] Marko Bertogna and Michele Cirinei. Response-Time Analysis for Globally Scheduled Symmetric Multiprocessor Platforms. In *Proc. of RTSS*, pages 149–160, 2007.

[BCA08] B.B. Brandenburg, J.M. Calandrino, and J.H. Anderson. On the Scalability of Real-Time Scheduling Algorithms on Multicore Platforms: A Case Study. *Proc. of RTSS*, pages 157 –169, 2008.

[BCB+08] B.B. Brandenburg, J.M. Calandrino, A. Block, H. Leontyev, and J.H. Anderson. Real-Time Synchronization on Multiprocessors: To Block or Not to Block, to Suspend or Spin? In *Proc. of RTAS*, pages 342 –353, 2008.

[BCGM99] Sanjoy Baruah, Deji Chen, Sergey Gorinsky, and Aloysius Mok. Generalized multiframe tasks. *Real-Time Syst.*, 17(1):5–22, July 1999.

[BCL05] M. Bertogna, M. Cirinei, and G. Lipari. New Schedulability Tests for Real-Time Task Sets Scheduled by Deadline Monotonic on Multiprocessors. In *Proc. of OPODIS*, pages 306–321, 2005.

[BCL09] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Schedulability Analysis of Global Scheduling Algorithms on Multiprocessor Platforms. *IEEE Tran. on Par. and Dist. Syst.*, 20(4):553–566, 2009.

[BCPV96] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[BDP96] A. Burns, R. Davis, and S. Punnekkat. Feasibility Analysis of Fault-Tolerant Real-Time Task Sets. In *Proc. of the ECRTS*, pages 522–527, 1996.

[BEL11] Stefan Bygde, Andreas Ermedahl, and Björn Lisper. An Efficient Algorithm for Parametric WCET Calculation. *Journal of Systems Architecture*, 57:614–624, May 2011.

[BF11] Sanjoy Baruah and Gerhard Fohler. Certification-Cognizant Time-Triggered Scheduling of Mixed-Criticality Systems. In *Proc. of RTSS*, pages 3–12, 2011.

[BFM97] A.A. Bertossi, A. Fusiello, and L.V. Mancini. Fault-tolerant deadline-monotonic algorithm for scheduling hard-real-time tasks. In *Parallel Processing Symposium, 1997. Proceedings., 11th International*, pages 133 –138, April 1997.

[BG03a] S. Baruah and J. Goossens. Rate-monotonic scheduling on uniform multiprocessors. *IEEE Transactions on Computers*, 52(7):966–970, 2003.

[BG03b] S. Baruah and J. Goossens. The Static-priority Scheduling of Periodic Task Systems upon Identical Multiprocessor Platforms. pages 427–432, 2003.

[BGJ06] V. Berten, J. Goossens, and E. Jeannot. A probabilistic approach for fault tolerant multiprocessor real-time scheduling. In *Proc. of IPDPS*, page 8, 2006.

[BLS10] S. Baruah, Haohan Li, and L. Stougie. Towards the Design of Certifiable Mixed-criticality Systems. In *Proc. of RTAS*, pages 13–22, 2010.

[BMR90] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proc. of the RTSS*, pages 182 –190, 1990.

[BMR99] A. A. Bertossi, L. V. Mancini, and F. Rossini. Fault-Tolerant Rate-Monotonic First-Fit Scheduling in Hard-Real-Time Systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(9):934–945, 1999.

[BPSW99] A. Burns, S. Punnekkat, L. Strigini, and D.R. Wright. Probabilistic scheduling guarantees for fault-tolerant real-time systems. In *Dependable Computing for Critical Applications*, pages 361–378, 1999.

[BRH90] S. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on One Processor. *Real-Time Systems.*, 2(4):301–324, 1990.

[BT83] J. A. Bannister and K. S. Trivedi. Task allocation in fault-tolerant distributed systems. *Acta Informatica*, 20:261–281, 1983.

[BV08] S. Baruah and S. Vestal. Schedulability Analysis of Sporadic Tasks with Multiple Criticality Specifications. In *Proc. of ECRTS*, pages 147–155, 2008.

[CB98] M. Caccamo and G. Buttazzo. Optimal scheduling for fault-tolerant and firm real-time systems . In *Proceedings of the IEEE Conference on Real-Time Computing Systems and Applications*, pages 223–231, 1998.

[CC89] H. Chetto and M. Chetto. Some Results of the Earliest Deadline Scheduling Algorithm. *IEEE Trans. on Soft. Engg.*, 15(10):1261–1269, 1989.

[CFH+04] J. Carpenter, S. Funk, P. Holman, J. H. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. *Handbook on Scheduling Algorithms, Methods, and Models*, 2004.

[CGG11] Liliana Cucu-Grosjean and Joël Goossens. Exact Schedulability Tests for Real-Time Scheduling of Periodic Tasks on Unrelated Multiprocessor Platforms. *Journal of Systems Architecture*, 57(5):561–569, 2011.

[CJD91] S.E. Chodrow, F. Jahanian, and M. Donner. Run-time monitoring of real-time systems. In *Proc. of RTSS*, pages 74 –83, 1991.

[CKR+12] Sudipta Chattopadhyay, Chong Lee Kee, Abhik Roychoudhury, Timon Kelter, Peter Marwedel, and Heiko Falk. A Unified WCET Analysis Framework for Multicore Platforms. In *Proc. of the RTAS*, pages 99–108, 2012.

[CLL90] J.-Y. Chung, J.W.S. Liu, and K.-J. Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Transactions on Computers*, 39(9):1156–1174, 1990.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[CMR92] A. Campbell, P. McDonald, and K. Ray. Single Event Upset Rates in Space. *IEEE Trans. on Nuclear Sci.*, 39(6):1828–1835, Dec 1992.

[CMS82] X. Castillo, S. R. McConnel, and D. P. Siewiorek. Derivation and Calibration of a Transient Error Reliability Model. *IEEE Transactions on Computers*, 31(7):658–671, 1982.

[CRD06] H. Cho, B. Ravindran, and E. Douglas. An Optimal Real-Time Scheduling Algorithm for Multiprocessors. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 101–110, 2006.

[CYKT07] Jian-Jia Chen, Chuan-Yue Yang, Tei-Wei Kuo, and Shau-Yin Tseng. Real-Time Task Replication for Fault Tolerance in Identical Multiprocessor Systems. In *Proc. of RTAS*, pages 249–258, 2007.

[DB09] Robert Davis and Alan Burns. Priority Assignment for Global Fixed Priority Pre-Emptive Scheduling in Multiprocessor Real-Time Systems. In *Proc. of RTSS*, pages 398–409, 2009.

[DB10] R.I. Davis and A. Burns. On optimal priority assignment for response time analysis of global fixed priority pre-emptive scheduling in multiprocessor hard real-time systems. Tech. report YCS-2010-451, University of York, Computer Science Dept., April 2010.

[DB11a] Robert Davis and Alan Burns. A Survey of Hard Real-Time Scheduling for Multiprocessor Systems. *ACM Computing Surveys*, 43(4):35:1–35:44, 2011.

[DB11b] Robert Davis and Alan Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Systems*, 47:1–40, 2011.

[Dha77] S. K. Dhall. Scheduling periodic-time - critical jobs on single processor and multiprocessor computing systems. *PhD Thesis, University of Illinois at Urbana-Champaign*, 1977.

[DL78] S. K. Dhall and C. L. Liu. On a Real-Time Scheduling Problem. *Operations Research*, 26(1):127–140, 1978.

[dNLR09] D. de Niz, K. Lakshmanan, and R. Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *Proc. of the RTSS*, pages 291 –300, 2009.

[DRRG10] François Dorin, Pascal Richard, Michaël Richard, and Joël Goossens. Schedulability and sensitivity analysis of multiple criticality tasks with fixed-priorities. *Real-Time Systems*, 46:305–331, 2010.

[EB08] Paul Emberson and Iain Bate. Extending a task allocation algorithm for graceful degradation of real-time distributed embedded systems. In *Proc. of the RTSS*, pages 270–279, 2008.

[EY12] P. Ekberg and Wang Yi. Bounding and shaping the demand of mixed-criticality sporadic tasks. In *Proc. of the ECRTS*, pages 135 –144, july 2012.

[FBB06] N. Fisher, S. Baruah, and T. P. Baker. The Partitioned Scheduling of Sporadic Tasks According to Static-Priorities. In *Proc. of ECRTS*, pages 118–127, 2006.

[GESY11a] Nan Guan, P. Ekberg, M. Stigge, and Wang Yi. Resource sharing protocols for real-time task graph systems. In *Proc. of the ECRTS*, pages 272 –281, 2011.

[GESY11b] Nan Guan, Pontus Ekberg, Martin Stigge, and Wang Yi. Effective and Efficient Scheduling of Certifiable Mixed-Criticality Sporadic Task Systems. In *Proc. of RTSS*, pages 13–23, 2011.

[GFB03] Joël Goossens, Shelby Funk, and Sanjoy Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Syst.*, 25(2-3):187–205, 2003.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[GLYY12] Nan Guan, Mingsong Lv, Wang Yi, and Ge Yu. WCET Analysis with MRU Caches: Challenging LRU for Predictability. *Proc. of RTAS*, pages 55–64, 2012.

[GMM94] S. Ghosh, R. Melhem, and D. Mosse. Fault-tolerant scheduling on a hard real-time multiprocessor system. In *Proc. of the Parallel Processing Symposium*, pages 775 –782, 1994.

[GMM95] S. Ghosh, R. Melhem, and D. Mossé. Enhancing Real-Time Schedules to Tolerate Transient Faults. In *Proc. of the RTSS*, pages 120–129, 1995.

[GMMS98] S. Ghosh, Rami Melhem, Daniel Mossé, and Joydeep Sen Sarma. Fault-Tolerant Rate-Monotonic Scheduling. *Real-Time Systems.*, 15(2):149–181, 1998.

[GSYY09] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. New Response Time Bounds for Fixed Priority Multiprocessor Scheduling. *Proc. of RTSS*, pages 387–397, 2009.

[GSYY10] Nan Guan, M. Stigge, Wang Yi, and Ge Yu. Fixed-Priority Multiprocessor Scheduling with Liu and Layland's Utilization Bound. In *Proc. of the RTAS*, pages 165 –174, 2010.

[HA05] Philip Holman and James H. Anderson. Adapting pfair scheduling for symmetric multiprocessors. *J. Embedded Comput.*, 1(4):543–564, December 2005.

[HL94] Rhan Ha and J.W.S. Liu. Validating timing constraints in multiprocessor and distributed real-time systems. In *Proc. of ICDCS*, pages 162 –171, 1994.

[HS89] D. Haban and K.G. Shin. Application of real-time monitoring to scheduling tasks with random execution times. In *Proc. of the RTSS*, pages 172 –181, 1989.

[HSW03] C.-C. Han, K. G. Shin, and J. Wu. A Fault-Tolerant Scheduling Algorithm for Real-Time Periodic Tasks with Possible Software Faults. *IEEE Trans. on Comp.*, 52(3):362–372, 2003.

[IRH86] R. K. Iyer, D. J. Rossetti, and M. C. Hsueh. Measurement and Modeling of Computer Reliability as Affected by System Activity. *ACM Trans. on Comp. Syst.*, 4(3):214–237, 1986.

[JHCS02] A. Jhumka, M. Hiller, V. Claesson, and N. Suri. On systematic design of globally consistent executable assertions in embedded software. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 75–84, 2002.

[Joh88] B. W. Johnson. *Design & analysis of fault tolerant digital systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[JP86] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, 1986.

[KLLS05a] K. Klonowska, H. Lennerstad, L. Lundberg, and C. Svahnberg. Optimal recovery schemes in fault tolerant distributed computing. *Acta Informatica.*, 41(6):341–365, 2005.

[KLLS05b] K. Klonowska, L. Lundberg, H. Lennerstad, and C. Svahnberg. Extended Golomb rulers as the new recovery schemes in distributed dependable computing. In *Proc. of the IPDPS*, page 8, 2005.

[KLR10] Junsung Kim, K. Lakshmanan, and R. Rajkumar. R-BATCH: Task Partitioning for Fault-tolerant Multiprocessor Real-Time Systems. In *Proc. of ICESS*, pages 1872 –1879, 2010.

[KSSF10] R. Kalla, B. Sinharoy, W.J. Starke, and M. Floyd. Power7: IBM's Next-Generation Server Processor. *Micro, IEEE*, 30(2):7 –15, 2010.

[KST11] Md Kamruzzaman, Steven Swanson, and Dean M. Tullsen. Inter-core prefetching for multicore processors using migrating helper threads. In *Proc. of ASPLOS*, pages 393–404, 2011.

[KY08] S. Kato and N. Yamasaki. Portioned static-priority scheduling on multiprocessors. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, pages 1–12, 2008.

[KY09] S. Kato and N. Yamasaki. Semi-Partitioned Scheduling of Sporadic Task Systems on Multiprocessors. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, pages 249–258, 2009.

[LB10a] Haohan Li and S. Baruah. An algorithm for scheduling certifiable mixed-criticality sporadic task systems. In *Proc. of RTSS*, pages 183–192, 2010.

[LB10b] Haohan Li and Sanjoy Baruah. Load-based schedulability analysis of certifiable mixed-criticality systems. In *Proc. of EMSOFT*, pages 99–108, 2010.

[LB12] Haohan Li and Sanjoy Baruah. Global mixed-criticality scheduling on multiprocessors. In *Proc. of ECRTS*, pages 166 – 175, 2012.

[LBOS95] J. Liebeherr, A. Burchard, Y. Oh, and S. H. Son. New Strategies for Assigning Real-Time Tasks to Multiprocessor Systems. *IEEE Trans. on Comp.*, 44(12):1429–1442, 1995.

[LDG04] J. M. López, J. L. Díaz, and D. F. García. Minimum and Maximum Utilization Bounds for Multiprocessor Rate Monotonic Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 15(7):642–653, 2004.

[LdNR11] K. Lakshmanan, D. de Niz, and R. Rajkumar. Mixed-criticality task synchronization in zero-slack scheduling. In *Proc. of RTAS*, pages 47 –56, 2011.

[LdNRM10] K. Lakshmanan, D. de Niz, R. Rajkumar, and G. Moreno. Resource allocation in distributed mixed-criticality cyber-physical systems. In *Proc. of the ICDCS*, pages 169 –178, 2010.

[LGDG03] J. M. López, M. García, J. L. Díaz, and D. F. García. Utilization Bounds for Multiprocessor Rate-Monotonic Scheduling. *Real-Time Systems*, 24(1):5–28, 2003.

[LL73] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LLMM99] Frank Liberato, Sylvain Lauzac, Rami Melhem, and Daniel Mossé. Fault Tolerant Real-Time Global Scheduling on Multiprocessors. *Proc. of ECRTS*, page 252, 1999.

[LMM98a] S. Lauzac, R. Melhem, and D. Mossé. An Efficient RMS Control and Its Application to Multiprocessor Scheduling. In *Proceedings of the International Parallel Processing Symposium*, pages 511–518, 1998.

[LMM98b] Sylvain Lauzac, Rami Melhem, and Daniel Mosse. Comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor. In *Euromicro Workshop on Real Time Systems*, pages 188–195, 1998.

[LMM00] F. Liberato, R. Melhem, and D. Mossé. Tolerance to Multiple Transient Faults for Aperiodic Tasks in Hard Real-Time Systems. *IEEE Trans. on Comp.*, 49(9):906–914, 2000.

[LNBCG11] Yue Lu, Thomas Nolte, Iain Bate, and Liliana Cucu-Grosjean. A New Way about using Statistical Analysis of Worst-Case Execution Times. *ACM SIGBED Review*, 8(2), 2011.

[LRL09] K. Lakshmanan, R. Rajkumar, and J. Lehoczky. Partitioned Fixed-Priority Preemptive Scheduling for Multi-core Processors. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, pages 239–248, 2009.

[LSD89] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 166–171, 1989.

[LSL+94] J.W.S. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung. Imprecise computations. *Proceedings of the IEEE*, 82(1):83–94, 1994.

[Lun02] L. Lundberg. Analyzing Fixed-Priority Global Multiprocessor Scheduling. In *Proc. of RTAS*, pages 145–153, 2002.

[LW82] J. Y. T. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. *Performance Evaluation*, 2:237–250, 1982.

[MAAMM00] P. Mejia-Alvarez, H. Aydin, D. Mossé, and R. Melhem. Scheduling optional computations in fault-tolerant real-time systems. In *Proc. of the RTCSA*, page 323, 2000.

[MAM99] P. Mejia-Alvarez and D. Mossé. A responsiveness approach for scheduling fault recovery in real-time systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pages 4–13, 1999.

[MBS07] A. Meixner, M.E. Bauer, and D.J. Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *Proc. of the Annual IEEE/ACM Int. Symp. on Microarchitecture*, pages 210–222, 2007.

[MCS91] H. Madeira, J. Camoes, and J. G. Silva. A watchdog processor for concurrent error detection in multiple processor systems. *Microprocessors and Microsystems*, 15(3):123–130, 1991.

[MD11] F. Many and D. Doose. Scheduling Analysis under Fault Bursts. In *Proc. of the RTAS*, pages 113 –122, 2011.

[MdALB03] G. M. de A. Lima and A. Burns. An optimal fixed-priority assignment algorithm for supporting fault-tolerant hard real-time systems. *IEEE Transactions on Computers*, 52(10):1332–1346, 2003.

[MEA+10] M.S. Mollison, J.P. Erickson, J.H. Anderson, S.K. Baruah, and J.A. Scoredos. Mixed-criticality real-time scheduling for multicore systems. In *Proc. of ICESS*, pages 1864–1871, 2010.

[MFC01] Aloysius K. Mok, Xiang (Alex) Feng, and Deji Chen. Resource partition for real-time systems. In *Proc. of the RTAS*, pages 75–, 2001.

[MM98] G. Manimaran and C. S. R. Murthy. A Fault-Tolerant Dynamic Scheduling Algorithm for Multiprocessor Real-Time Systems and Its Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 9(11):1137–1152, 1998.

[NLR09] Dionisio de Niz, Karthik Lakshmanan, and Ragunathan Rajkumar. On the Scheduling of Mixed-Criticality Real-Time Task Sets. In *Proc. of RTSS*, pages 291–300, 2009.

[NSBS09] T. Nolte, Insik Shin, M. Behnam, and M. Sjodin. A Synchronization Protocol for Temporal Isolation of Software Components in Vehicular Systems. *IEEE Transactions on Industrial Informatics*, 5(4):375 –387, nov. 2009.

[OB98] D.-I. Oh and T. P. Baker. Utilization Bounds for N-Processor Rate Monotone Scheduling with Static Processor Assignment. *Real-Time Systems*, 15(2):183–192, 1998.

[OS94] Y. Oh and S. H. Son. Enhancing fault-tolerance in rate-monotonic scheduling. *Real-Time Systems.*, 7(3):315–329, 1994.

[OS95a] Y. Oh and S. H. Son. A Processor-Efficient Scheme for Supporting Fault-Tolerance in Rate-Monotonic Scheduling. Technical report, University of Virginia, 1995.

[OS95b] Yingfeng Oh and Sang H. Son. Allocating fixed-priority periodic tasks on multiprocessor systems. *Real-Time Systems.*, 9(3):207–239, 1995.

[PBD01] S. Punnekkat, A. Burns, and R. Davis. Analysis of Checkpointing for Real-Time Systems. *Real-Time Systems.*, 20(1):83–102, 2001.

[PJ10] R.M. Pathan and J. Jonsson. Load regulating algorithm for static-priority task scheduling on multiprocessors. In *Proc. of the IPDPS*, pages 1 –12, 2010.

[PLHE09] S.M. Petters, M. Lawitzky, R. Heffernan, and K. Elphinstone. Towards real multi-criticality scheduling. In *Proc. of RTCSA*, pages 155 – 164, 2009.

[PM98] M. Pandya and M. Malek. Minimum Achievable Utilization for Fault-Tolerant Processing of Periodic Tasks. *IEEE Trans. on Comp.*, 47(10):1102–1112, 1998.

[PMCR08] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu. Hardware Runtime Monitoring for Dependable COTS-Based Real-Time Embedded Systems. In *Proc of the RTSS*, pages 481 –491, 30 2008-dec. 3 2008.

[PMN+09] Rodolfo Pellizzoni, Patrick Meredith, Min-Young Nam, Mu Sun, Marco Caccamo, and Lui Sha. Handling mixed-criticality in soc-based real-time embedded systems. In *Proc. of EMSOFT*, pages 235–244, 2009.

[Pra07] Fault-tolerant systems. Morgan Kaufmann, 2007.

[RRJ92] S.C.V. Raju, R. Rajkumar, and F. Jahanian. Monitoring timing constraints in distributed real-time systems. In *Proc. of the RTSS*, pages 57 –67, 1992.

[SABR04] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The Impact of Technology Scaling on Lifetime Reliability. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 177–186, 2004.

[Sch84] Fred B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Trans. Comput. Syst.*, 2(2):145–154, 1984.

[SEGY11] Martin Stigge, Pontus Ekberg, Nan Guan, and Wang Yi. The digraph real-time task model. In *Proc. of the RTAS*, pages 71–80, 2011.

[SG90] Lui Sha and John B. Goodenough. Real-time scheduling theory and ada. *Computer*, 23(4):53–62, 1990.

[SGTG12] F. Santy, L. George, P. Thierry, and J. Goossens. Relaxing Mixed-Criticality Scheduling Strictness for Task Sets Scheduled with FP. In *Proc. pf the ECRTS*, pages 155 –165, 2012.

[SKK+02] P. Shivakumar, M. Kistler, S.W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proc. of the DSN*, pages 389 – 398, 2002.

[SKK+08] P. N. Sanda, J. W. Kellington, P. Kudva, R. Kalla, R. B. McBeth, J. Ackaret, R. Lockwood, J. Schumann, and C. R. Jones. Soft-error resilience of the IBM POWER6 processor. *IBM J. Res. Dev.*, 52(3):275–284, 2008.

[SKM+78] D.P. Siewiorek, V. Kini, H. Mashburn, S. McConnel, and M. Tsao. A case study of C.mmp, Cm*, and C.vmp: Part I–Experiences with fault tolerance in multiprocessor systems. *Proceedings of the IEEE*, 66(10):1 –1199, 1978.

[SLR86] L. Sha, J. P. Lehoczky, and R. Rajkumar. Solutions for Some Practical Problems in Prioritized Preemptive Scheduling. In *Proc. of RTSS*, pages 181–191, 1986.

[SMR11] Abhik Sarkar, Frank Mueller, and Harini Ramaprasad. Predictable task migration for locked caches in multi-core systems. In *Proc. of LCTES*, pages 131–140, 2011.

[SMRM09] Abhik Sarkar, Frank Mueller, Harini Ramaprasad, and Sibin Mohan. Push-assisted migration of real-time tasks in multi-core processors. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 80–89, 2009.

[SS83] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1:222–238, 1983.

[SS94] M. Spuri and J.A. Stankovic. How to Integrate Precedence Constraints and Shared Resources in Real-Time Scheduling. *IEEE Transactions on Computers*, 43:1407–1412, 1994.

[SS99] P. Sinha and N. Suri. On the use of formal techniques for analyzing dependable real-time protocols. In *Proc. of RTSS*, pages 126 –135, 1999.

[SSO05] R. M. Santos, J. Santos, and J. D. Orozco. A Least Upper Bound on the Fault Tolerance of Real-Time Systems. *Jour. of Sys. and Soft.*, 78(1):47–55, 2005.

[SUSO04] R. M. Santos, J. Urriza, J. Santos, and J. D. Orozco. New Methods for Redistributing Slack Time in Real-Time Systems: Applications and Comparative Evaluations. *Jour. of Sys. and Soft.*, 69(1-2):115–128, 2004.

[SXLC11a] A. Saifullah, You Xu, Chenyang Lu, and Yixin Chen. End-to-End Delay Analysis for Fixed Priority Scheduling in WirelessHART Networks. In *Proc. of RTAS*, pages 13 –22, 2011.

[SXLC11b] Abusayeed Saifullah, You Xu, Chenyang Lu, and Yixin Chen. Priority Assignment for Real-time Flows in WirelessHART Networks. In *Proc. of ECRTS*, pages 33–44, 2011.

[TKK95] T. Tsuchiya, Y. Kakuda, and T. Kikuno. Fault-tolerant scheduling algorithm for distributed real-time systems. In *Proceedings of the Workshop on Parallel and Distributed Real-Time Systems*, page 99, 1995.

[TSP11] D. Tamas-Selicean and P. Pop. Design Optimization of Mixed-Criticality Real-Time Applications on Cost-Constrained Partitioned Architectures. In *Proc. of RTSS*, pages 24 –33, 2011.

[Ves07] S. Vestal. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. In *Proc. of RTSS*, pages 239–243, 2007.

[WEMR04] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor. In *Proceedings of the annual international symposium on Computer architecture*, pages 264–275, 2004.

[WHA] WirelessHART Specification, www.hartcomm.org, 2007.

[XP00] Jia Xu and David Lorge Parnas. Priority scheduling versus pre-run-time scheduling. *Real-Time Syst.*, 18(1):7–23, January 2000.

[YKS11] Man-Ki Yoon, Jung-Eun Kim, and Lui Sha. Optimizing tunable wcet with shared resource allocation and arbitration in hard real-time multicore systems. In *Proc. of the RTSS*, pages 227 –238, 2011.

[YYP+12] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory Access Control in Multiprocessor for Real-time Systems with Mixed Criticality. In *ECRTS*, pages 299 – 308, 2012.

[ZMM03] Dakai Zhu, Daniel Mossé, and Rami Melhem. Multiple-Resource Periodic Scheduling Problem: how much fairness is necessary? *Proceedings of the IEEE Real-Time Systems Symposium*, pages 142–151, 2003.

[ZQQ11] Xiaomin Zhu, Xiao Qin, and Meikang Qiu. QoS-Aware Fault-Tolerant Scheduling for Real-Time Tasks on Heterogeneous Clusters. *IEEE Trans. Comput.*, 60:800–812, 2011.

A

Proofs of Theorems and Lemmas

Lemma 5.3 (from Chapter 5)**.** *Consider a, b, x, c and d such that $0 \leq a \leq b \leq x \leq c \leq d \leq \frac{m}{2m-1}$ for any integer $m > 0$. The following two inequalities hold:*

$$min\{F_m(b),F_m(c)\} \leq F_m(x) \tag{5.4}$$

$$min\{F_m(a),F_m(d)\} \leq min\{F_m(b),F_m(c)\} \tag{5.5}$$

Proof. To show that Eq. (5.4) holds we will show that, the function $F_m(x) = \frac{m(1-x)}{2-x} + x$ achieves its absolute minimum at one of the end-points in $[b, c]$, where $b \leq x \leq c$, for any given m. Thus, the minimum between $F_m(b)$ and $F_m(c)$ is the absolute minimum of $F_m(x)$, and consequently Eq. (5.4) holds.

The first derivative of function $F_m(x)$ with respect to x is $F'_m(x) = 1 - \frac{m}{(2-x)^2}$. By setting $F'_m(x) = 0$, we have $x = (2 \pm \sqrt{m})$. For any value of $m > 0$, the point $x = (2 + \sqrt{m})$ is outside of (b, c) since $c \leq \frac{m}{2m-1} \leq 1$ for $m > 0$. Moreover, the point $x = (2 - \sqrt{m})$ is outside of $(0, \frac{m}{2m-1})$ for both $m = 1$ and $m \geq 4$. Consequently, $x = (2 - \sqrt{m})$ is also outside of (b, c) because (b, c) is entirely contained within $(0, \frac{m}{2m-1})$ for $m = 1$ and $m \geq 4$. So, the only possible x values satisfying both $x = (2 - \sqrt{m})$ and $F'_m(x) = 0$ are $x = (2 - \sqrt{2})$ and $x = (2 - \sqrt{3})$ for $m = 2$ and $m = 3$, respectively (called the *stationary points*). Since there is no stationary point of $F_m(x)$ within (b, c) for $m = 1$ or $m \geq 4$, the absolute minimum of $F_m(x)$ occurs at one of the end points of $[b, c]$ for $m = 1$ and $m \geq 4$. So, only the cases where $m = 2$ and $m = 3$ need to be considered.

Now for $m = 2$, if the point $x = (2 - \sqrt{2})$ is outside of (b, c), then the absolute minimum of $F_2(x)$ occurs at one of the endpoints of $[b, c]$. Otherwise, if the point

$x = (2 - \sqrt{2})$ is within (b, c), then the absolute minimum occurs at one of the three points $x = a$, $x = (2 - \sqrt{2})$, or $x = b$. The function $F_2(x)$ is increasing within $[b, 2 - \sqrt{2})$ since $F_2'(x) = 1 - \frac{2}{(2-x)^2} > 0$ within $(b, 2 - \sqrt{2})$ and $F_2(x)$ is decreasing within $(2 - \sqrt{2}, c]$ since $F_2'(x) = 1 - \frac{2}{(2-x)^2} < 0$ within $(2 - \sqrt{2}, c)$. Therefore, the function $F_2(x)$ has its absolute maximum at $x = (2 - \sqrt{2})$. Thus, the absolute *minimum* of $F_2(x)$ occurs at one of the end points of $[b, c]$.

Similarly for $m = 3$, if the point $x = (2 - \sqrt{3})$ is outside of (b, c), then the absolute minimum of $F_3(x)$ occurs at one of the endpoints of $[b, c]$. Otherwise, if the point $x = (2 - \sqrt{3})$ is within (b, c), then the absolute minimum occurs at one of the three points $x = a$, $x = (2 - \sqrt{3})$, or $x = b$. The function $F_3(x)$ is increasing within $[b, 2 - \sqrt{3})$ since $F_3'(x) = 1 - \frac{3}{(2-x)^2} > 0$ within $(b, 2 - \sqrt{3})$ and $F_3(x)$ is decreasing within $(2 - \sqrt{3}, c]$ since $F_3'(x) = 1 - \frac{3}{(2-x)^2} < 0$ within $(2 - \sqrt{3}, c)$. Therefore, the function $F_3(x)$ has its absolute maximum at $x = (2 - \sqrt{3})$. Consequently, the absolute *minimum* of $F_3(x)$ occurs at one of the end points of $[b, c]$.

Since the function $F_m(x)$ has its minimum at one of the end points of $[b, c]$ for any m, it can be concluded that if x is within $[b, c]$ then $F_m(x)$ is not less than the minimum between $F_m(b)$ and $F_m(c)$. Therefore, Eq. (5.4) holds.

Since according to the premise $a \leq b \leq d$ and $a \leq c \leq d$, it follows from Eq. (5.4) that $min\{F_m(a), F_m(d)\} \leq F_m(b)$ and $min\{F_m(a), F_m(d)\} \leq F_m(c)$ which imply that $min\{F_m(a), F_m(d)\} \leq min\{F_m(b), F_m(c)\}$ holds. □

Lemma 5.4 (from Chapter 5)**.** *Consider the sporadic task system Γ^k that is special on m processors. The following inequality holds for $m \geq 1$*

$$min\{F_m(\delta_{min}^k)\ ,\ F_m(\delta_{max}^k)\ \} \ \leq \ \frac{m^2}{2m-1} \tag{5.6}$$

Proof. We show that the inequality in Eq. (5.6) holds by considering four different cases: Case (i) $m = 1$, Case (ii) $m = 2$, Case (iii) $m = 3$, and Case (iv) $m \geq 4$. Remember that, according to Property 1 of special task set Γ^k, we have $\delta_{max}^k \leq \frac{m}{2m-1}$.

Case (i) $m = 1$: The function $F_1(x)$ is increasing within $[0, 1]$ since $F_1'(x) = 1 - \frac{1}{(2-x)^2} > 0$ within $(0, 1)$. Thus, the maximum of $F_1(x)$ within $[0, 1]$ occurs at $x = 1$, and $F_1(1) = 1$. Note that δ_{min}^k and δ_{max}^k are within $[0, 1]$ since each task's density is assumed to be within $[0, 1]$. Therefore, we have $min\{F_1(\delta_{min}^k), F_1(\delta_{max}^k)\} \leq F_1(1) = 1$ for $m = 1$. Because $\frac{m^2}{2m-1} = 1$ for $m = 1$, we have $min\{F_m(\delta_{min}^k), F_m(\delta_{max}^k)\} \leq \frac{m^2}{2m-1}$.

Case (ii) $m = 2$: Since $\frac{m}{2m-1} = \frac{2}{3}$ for $m = 2$ and $\delta_{max}^k \leq \frac{m}{2m-1}$, both δ_{min}^k and δ_{max}^k of are within $[0, \frac{2}{3}]$. The function $F_2(x)$ is increasing within $[0, 2 - \sqrt{2}]$ since $F_2'(x) = 1 - \frac{2}{(2-x)^2} > 0$ within $(0, 2 - \sqrt{2})$ and the function $F_2(x)$ is decreasing within $[2 - \sqrt{2}, \frac{2}{3}]$ since $F_2'(x) = 1 - \frac{2}{(2-x)^2} < 0$ within $(2 - \sqrt{2}, \frac{2}{3})$. Therefore, the function $F_2(x)$ has its maximum at $x = (2 - \sqrt{2})$ within $[0, \frac{2}{3}]$, and $F_2(2 - \sqrt{2}) = 2(2 - \sqrt{2})$.

Consequently, *min*$\{F_m(\delta^k_{min})$, $F_m(\delta^k_{max})\} \leq F_2(2-\sqrt{2})$. Since $F_2(2-\sqrt{2}) = 2(2-\sqrt{2}) \leq \frac{4}{3} = \frac{m^2}{2m-1}$ for $m=2$, we have *min*$\{F_m(\delta^k_{min})$,$F_m(\delta^k_{max}\})\} \leq \frac{m^2}{2m-1}$.

Case (iii) $m=3$: Since $\frac{m}{2m-1} = \frac{3}{5}$ for $m=3$ and $\delta^k_{max} \leq \frac{m}{2m-1}$, both δ^k_{min} and δ^k_{max} of are within $[0, \frac{3}{5}]$. The function $F_3(x)$ is increasing within $[0, 2-\sqrt{3}]$ since $F_3'(x) = 1 - \frac{3}{(2-x)^2} > 0$ within $(0, 2-\sqrt{3})$ and the function $F_3(x)$ is decreasing within $[2-\sqrt{3}, \frac{3}{5}]$ since $F_3'(x) = 1 - \frac{3}{(2-x)^2} < 0$ within $(2-\sqrt{3}, \frac{3}{5})$. Therefore, the function $F_3(x)$ has its maximum at $x = (2-\sqrt{3})$ within $[0, \frac{3}{5}]$, and $F_3(2-\sqrt{3}) = (5-2\sqrt{3})$. Consequently, *min*$\{F_m(\delta^k_{min})$, $F_m(\delta^k_{max})\} \leq F_3(2-\sqrt{3})$. Since $F_3(2-\sqrt{3}) = (5-2\sqrt{3}) \leq \frac{9}{5} = \frac{m^2}{2m-1}$ for $m=3$, we have *min*$\{F_m(\delta^k_{min})$,$F_m(\delta^k_{max}\})\} \leq \frac{m^2}{2m-1}$.

Case (iv) $m \geq 4$: The function $q(m) = \frac{m}{2m-1}$ is decreasing for $m \geq 4$ because $q'(m) = \frac{-1}{(2m-1)^2} < 0$ for $m \geq 4$. Therefore, $\frac{m}{2m-1} \leq \frac{4}{7}$ for $m \geq 4$, and both δ^k_{min} and δ^k_{max} are within $[0, \frac{4}{7}]$. The function $F_m(x)$ is decreasing within $[0, \frac{4}{7}]$ for $0 \leq x \leq \frac{4}{7}$ since $F_m'(x)$ =1 $- \frac{m}{(2-x)^2} < 0$ within $(0, \frac{4}{7})$ for $m \geq 4$. Thus, the maximum of $F_m(x)$ occurs at $x=0$, and $F_m(0) = \frac{m}{2}$. Therefore, *min*$\{F_m(\delta^k_{min})$, $F_m(\delta^k_{max})\} \leq F_m(0)$. Since $F_m(0) = \frac{m}{2} < \frac{m^2}{2m-1}$ for $m \geq 4$, we have *min*$\{F_m(\delta^k_{min})$, $F_m(\delta^k_{max})\} \leq \frac{m^2}{2m-1}$.

It is proved for all the cases that if Γ^k is special on m processors, then the inequality in Eq. (5.6) holds. □

Lemma A.1. *The following inequality holds for* $m \geq m' \geq 1$.

$$B(m) \leq \frac{m}{2m-1} \leq \frac{m'}{2m'-1} \tag{A.1}$$

where $B(m)$ *is the function defined in Eq.* (5.12)*.*

Proof. We prove this Lemma considering three cases: Case (i) $m=1$, Case (ii) $m=2$, and Case (iii) $m \geq 3$.

Case (i) $m=1$**:** For this case, we have $m = m' = 1$ since $m \geq m' \geq 1$. Therefore, $\frac{m}{2m-1} = \frac{m'}{2m'-1} = 1$ for $m = m' = 1$. From Eq. (5.12), we have $B(m) = B(1) = 1$ for $m=1$. Therefore, Eq. (A.1) holds.

Case (ii) $m=2$**:** Using Eq. (5.12), we have $B(2) = (2-\sqrt{2})$ for $m=2$. And $\frac{m}{2m-1} = \frac{2}{3}$ for $m=2$. Because $(2-\sqrt{2}) < \frac{2}{3}$, we have $B(m) < \frac{m}{2m-1}$ for $m=2$. The function $q(x) = \frac{x}{2x-1}$ is decreasing for $x \geq 1$ because $q'(x) = \frac{-1}{(2x-1)^2} < 0$ for $x > 1$. Thus, we have $\frac{m}{2m-1} \leq \frac{m'}{2m'-1}$ for $m \geq m'$. Consequently, $B(m) < \frac{m}{2m-1} \leq \frac{m'}{2m'-1}$. Therefore, Eq. (A.1) holds.

Case (iii) $m \geq 3$: The following inequality in Eq (A.2) holds for any m such that $m \geq 3$.

$$\begin{aligned} & 0 \leq m^2 - 4m + 3 \qquad (A.2) \\ \equiv\ & 4m^2 - 4m + 1 \leq 5m^2 - 8m + 4 \\ \equiv\ & 2m - 1 \leq \sqrt{5m^2 - 8m + 4} \end{aligned}$$

$$\begin{aligned} \equiv\ & 3m - 2 - \sqrt{5m^2 - 8m + 4} \leq m - 1 \\ \equiv\ & \frac{3m - 2 - \sqrt{5m^2 - 8m + 4}}{2m - 2} \leq \frac{1}{2} \\ \Rightarrow\ & \left[\text{since } B(m) = \frac{3m - 2 - \sqrt{5m^2 - 8m + 4}}{2m - 2} \text{ according to Eq. (5.12) for } m \geq 3\right] \\ \equiv\ & B(m) \leq \frac{1}{2} \\ & \left[\text{since } \frac{1}{2} \leq \frac{m}{2m-1} \leq \frac{m'}{2m'-1} \text{ for } m \geq m' \geq 1\right] \\ \Rightarrow\ & B(m) \leq \frac{m}{2m-1} \leq \frac{m'}{2m'-1} \end{aligned}$$

Therefore, Eq. (A.1) holds for all the cases. □

Lemma A.2. *Let m and m' be two integers such that $m \geq m' \geq 1$. The following inequality in Eq.* (A.3) *holds*

$$B(m) \leq B(m') \qquad (A.3)$$

where $B(m)$ is the function defined in Eq. (5.12)*.*

Proof. For $m \geq 2$, the first derivative of $B(m) = \frac{3m-2-\sqrt{5m^2-8m+4}}{2m-2}$ is $B'(m) = \frac{-2(\sqrt{5m^2-8m+4}-m)}{(\sqrt{5m^2-8m+4})(2m-2)^2}$. Note that we have $B'(m) < 0$ because $\sqrt{5m^2 - 8m + 4} > m$ for $m \geq 2$. So, $B(m)$ is *decreasing* for $m \geq 2$. Thus, the maximum of $B(m)$ occurs at $m = 2$ whenever $m \geq 2$, and $B(2) = (2 - \sqrt{2})$. From Eq. (5.12), we have $B(1) = 1$. Since $1 > (2 - \sqrt{2})$, we have $B(1) > B(m)$ for any $m \geq 2$.

If $m = m'$, then Eq. (A.3) trivially holds. So, to prove this Lemma, we consider where $m > m'$. Note that $m \geq 2$ whenever $m > m'$ because $m' \geq 1$.

Now, if $m' = 1$, then $B(m') > B(m)$. This is because $B(1) > B(m)$ for any $m \geq 2$. Otherwise, if $m' > 1$, then $m > 2$ since we consider $m > m'$. Because the function $B(m)$ is decreasing for $m \geq 2$, we have $B(m') > B(m)$ where $m > m'$. Therefore, if $m \geq m' \geq 1$, we have $B(m) \leq B(m')$. □

Theorem 5.5 (from Chapter 5)**.** *An implicit-deadline sporadic task set* Γ *is schedulable using global* FP *scheduling under* SM-US$[\sqrt{2}-1]$ *priority assignment policy, if the following condition, for* $m \geq 2$*, holds:*

$$U^n \leq m \cdot (\sqrt{2}-1)$$

where $u_i \leq (1-\frac{1}{\sqrt{2}})$ *or* $u_i > (\sqrt{2}-1)$ *for each* $\tau_i \in \Gamma$.

Proof. Given the taskset Γ and the number of processors m, the two subsets Γ_L and Γ_H such that $\Gamma = \Gamma_L \cup \Gamma_H$ based on the threshold density or utilization $\delta_{ts} = (\sqrt{2}-1)$ can be determined. We will show that if the total utilization $U^n \leq m \cdot (\sqrt{2}-1)$, then the two general conditions **C1** and **C2** of Lemma 5.6 hold; which guarantees the schedulability of Γ using global FP scheduling if no task's utilization is within the range $(1-\frac{1}{\sqrt{2}}, \sqrt{2}-1]$.

Each task in Γ_H has utilization greater than $(\sqrt{2}-1)$ for the SM-US$[\sqrt{2}-1]$ policy. Since the total utilization of taskset Γ is not greater than $(\sqrt{2}-1)m$ according to the premise, the number of tasks that are given the highest priority is less than m (**C1** holds).

To show that **C2** of Lemma 5.6 holds, we have to show that Γ_L is special on m' processors where $m' = (m - |\Gamma_H|)$. Let UL be the total utilization of all the tasks in Γ_L. Also let u_{maxL} and u_{minL} be the maximum and minimum utilization of any task in set Γ_L, respectively. To show that Γ_L is special on m' processors, we show that Property 1 and Property 2 (given in Definition 5.1) of special taskset are satisfied. In other words, we have to show that the following two inequalities hold.

Property 1 $\quad u_{maxL} \leq \dfrac{m'}{2m'-1}$

Property 2 $\quad \mathsf{UL} \;\leq\; min\{F_{m'}(u_{minL}), F_{m'}(u_{maxL})\}$

(Property 1 holds for Γ_L) Since $u_i \leq (1-1/\sqrt{2})$ or $u_i > (\sqrt{2}-1)$ for each task $\tau_i \in \Gamma$, no task in Γ_L has utilization greater than $(1-1/\sqrt{2})$ for the SM-US$[\sqrt{2}-1]$ policy. So, $u_{maxL} \leq (1-1/\sqrt{2})$. Note that $(1-1/\sqrt{2}) \leq \frac{m'}{2m'-1}$ for any integer $m' > 0$. Consequently, $u_{maxL} \leq \frac{m'}{2m'-1}$, and thus, Property 1 is satisfied.

(Property 2 holds for Γ_L) The total utilization of the tasks in Γ_H is greater than $(|\Gamma_H| \cdot (\sqrt{2}-1))$ because each task in Γ_H has utilization greater than $(\sqrt{2}-1)$ for the SM-US$[\sqrt{2}-1]$ policy. Since the total utilization of Γ is not greater than $m \cdot (\sqrt{2}-1)$ according to the premise, the total utilization of the tasks in Γ_L is at most $m' \cdot (\sqrt{2}-1)$ where $m' = (m - |\Gamma_H|)$. Therefore, Eq. (A.4) holds.

$$\mathsf{UL} \leq m' \cdot (\sqrt{2}-1)\} \tag{A.4}$$

Since $0 \leq u_{minL} \leq u_{maxL} \leq (1 - 1/\sqrt{2}) \leq \frac{m'}{2m'-1}$, from Eq. (5.5), we have

$$min\{F_{m'}(0)\ ,\ F_{m'}(1 - 1/\sqrt{2})\} \leq min\{F_{m'}(u_{minL}),F_{m'}(u_{maxL})\} \tag{A.5}$$

From the function definition given in Eq. (5.3), we have

$$F_{m'}(0) = \frac{m'(1-0)}{2-0} + 0 = m'/2 = m' \cdot 1/2 \tag{A.6}$$

$$\begin{aligned} F_{m'}(1 - 1/\sqrt{2}) &= \frac{m'(1-(1-1/\sqrt{2}))}{2-(1-1/\sqrt{2})} + (1 - 1/\sqrt{2}) \\ &= m'(\sqrt{2}-1) + (1 - 1/\sqrt{2}) > m' \cdot (\sqrt{2}-1) \end{aligned} \tag{A.7}$$

It follows from Eq. (A.6) and Eq. (A.7) that

$$min\{F_{m'}(0), F_{m'}(1 - 1/\sqrt{2})\} \geq m' \cdot (\sqrt{2}-1) \tag{A.8}$$

Thus, it now follows from Eq. (A.4) and Eq. (A.8) that

$$\mathsf{UL} \leq min\{F_{m'}(0), F_{m'}(1 - 1/\sqrt{2})\ \} \tag{A.9}$$

Finally, from Eq. (A.5) and Eq. (A.9), we have

$$\mathsf{UL} \leq min\{F_{m'}(u_{minL}), F_{m'}(u_{maxL})\ \} \tag{A.10}$$

Therefore, Property 2 is satisfied for task set Γ_L. Consequently, Γ_L is special on m' processors (i.e., **C2** holds). □

Theorem 6.3 (from Chapter 6)**.** *If task set* Γ *is schedulable using the* `H-ODA-LC` *test, then* Γ *is also schedulable using the* `IA-DA` *test, and not conversely.*

Proof. Assume a contradiction that task set Γ does not pass the `IA-DA` test but passes the `H-ODA-LC` test. Note that `IA-DA` test *cannot* fail to assign priorities between priority levels $(n-m+1)$ and n because the `IA-DA` algorithm in Figure 6.4 assigns these m highest priority levels in line 12–13 and returns "schedulable" in line 14. Therefore, the `IA-DA` test can fail to assign priority only at some priority level between 1 and $(n-m)$.

Let the `IA-DA` test *first* fails to assign priority at some priority level `PL`, where $1 \leq \mathsf{PL} \leq (n-m)$. Thus, when `IA-DA` test fails at priority level `PL`, there are total `(PL-1)` tasks that are already assigned fixed priorities and there are total $(n - \mathsf{PL} + 1)$ priority-unassigned tasks. Consequently, the *minimum* number of priority-unassigned tasks when `IA-DA` fails is $(m+1)$ since $1 \leq \mathsf{PL} \leq (n-m)$. Let `F` denotes the set of all priority-unassigned tasks when `IA-DA` fails. Note that $|\mathsf{F}| \geq (m+1)$.

Remember that the `H-ODA-LC` test assigns the highest fixed priority to the m' highest-density tasks and the remaining $(n-m')$ lowest-density tasks are assigned pri-

orities based on the ODA-LC test for some m', where $0 \leq m' < m$. Since Γ passes the H-ODA-LC test, there are $(n - m')$ lowest-density tasks that are successfully assigned priorities using the ODA-LC test for some m', $0 \leq m' < m$. In other words, each of the $(n - m')$ lowest-density tasks passes the DA-LC test (because the ODA-LC test essentially applies the DA-LC test in algorithm OPA in Figure 6.1). Let P denotes the set of these $(n - m')$ lowest-density tasks. Note that $|\mathsf{P}| \geq (n - m + 1)$ since $0 \leq m' < m$.

Because $|\mathsf{F}| + |\mathsf{P}| \geq (m + 1) + (n - m + 1) = (n + 2)$ and $|\Gamma| = n$, there are at least two tasks that are common to both sets F and P. Let τ_x be such a common task where task $\tau_x \in (F \cap P)$. Without loss of generality assume that each task in set $((F \cap P) - \{\tau_x\})$ has higher priority than that of task τ_x for the priorities assigned by the H-ODA-LC test. Therefore, each of the tasks in $(\mathsf{F} - \{\tau_x\})$ is assigned higher priorities than that of task τ_x according to the priorities assigned by the H-ODA-LC test. In other words, if ϕ is the set of tasks that are assigned higher priorities than task τ_x according to the priorities assigned by the H-ODA-LC test, then $(\mathsf{F} - \{\tau_x\}) \subseteq \phi$.

Since $\tau_x \in \mathsf{P}$, task τ_x passes DA-LC test when assigning the priority using the H-ODA-LC test. Note that set ϕ includes the m' highest-density tasks that are separated and assigned the highest fixed-priority in H-ODA-LC test. If task τ_x passes the DA-LC test, where m' highest-density tasks from set ϕ are separated, then according to Lemma 6.1, task τ_x must pass the DA-LC test by separating m' tasks using algorithm Select(ϕ, m', τ_x, D_x) from set ϕ. Consequently, task τ_x must pass the DA-LC test by separating m' or lower number of tasks from set $(F - \{\tau_x\})$ using the Select algorithm since $(F - \{\tau_x\}) \subseteq \phi$. Therefore, the IA-DA test that uses the Select algorithm for separation of the tasks can not fail to assign priority to task τ_x at priority level PL if Γ passes the H-ODA-LC test. Therefore, any task set that passes the H-ODA-LC test also passes the IA-DA test. The task set in Example 6.2 passes the IA-DA test but not the H-ODA-LC test. Therefore, IA-DA test dominates the state-of-the-art H-ODA-LC test. □

Lemma 9.2 (form Chapter 9)**.** *The net increase in workload due to any shift of the problem window in Figure 9.4 is bounded by* $C_k^{HI} - C_k^{LO}$.

Proof. This Lemma is proved by considering any possible shift of the problem window for the reference pattern in Figure 9.4 both in (i) leftward, and (ii) rightward directions for α time units, $0 \leq \alpha \leq T_k$. Shifting the problem window by exactly T_k time units in any direction results in the same release pattern as in Figure 9.4. For ease of readability, Figure 9.4 is presented here again in Figure A.1.

Leftward shift: Due to the leftward shift of the problem window in the reference pattern, the workload in the shifted window may *increase* in two of the following ways:

- First, the job $J_k^{(y-1)}$ that was executing for C_k^{LO} time units in the reference pattern may now experience the criticality-switch in the shifted window. Thus, the workload of job $J_k^{(y-1)}$ may now increase by $(C_k^{\mathsf{HI}} - C_k^{\mathsf{LO}})$ time units within the shifted window.

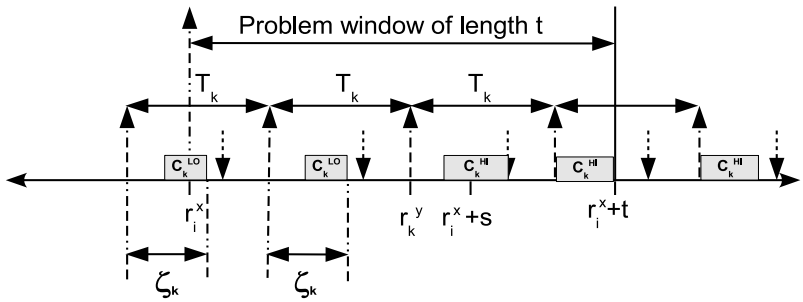

Figure A.1: *The reference pattern. Each job released before r_k^y finishes $(D_k - \zeta_k)$ time units earlier than its deadline in the reference pattern.*

- Second, *new* workload may enter into the shifted window from the left-hand side of the window. Note that any job that is released before job J_k^y executes for at most $C_k^{\mathtt{LO}}$ time units. Thus, the amount of new workload that may enter from the left-hand side of the problem window is bounded by $C_k^{\mathtt{LO}}$.

Consequently, a (pessimistic) upper bound on the total increase in workload within the shifted window is $C_k^{\mathtt{HI}}$ time units. However, workload in the reference pattern may also decrease from the right-hand side of the problem window.

Shifting the problem window left by α time units, where $0 \leq \alpha \leq C_k^{\mathtt{LO}}$, the workload in the shifted window is *decreased* by α time units from the right-hand side. In such case, new workload that may enter into the shifted window from the left-hand side is at most α. Because the execution time of job $J_k^{(y-1)}$ may now be increased by $(C_k^{\mathtt{HI}} - C_k^{\mathtt{LO}})$ time units, the ***net*** increase in workload within the shifted problem window is at most $(C_k^{\mathtt{HI}} - C_k^{\mathtt{LO}})$, whenever $0 \leq \alpha \leq C_k^{\mathtt{LO}}$.

Shifting the problem window left by α time units, where $C_k^{\mathtt{LO}} < \alpha \leq T_k$, the workload in the shifted window decreases by at least $C_k^{\mathtt{LO}}$ time units from the right-hand side of the window. Because an upper bound on the total increase in workload in the shifted window is $C_k^{\mathtt{HI}}$, the maximum ***net*** increase in workload is bounded by $(C_k^{\mathtt{HI}} - C_k^{\mathtt{LO}})$ whenever $C_k^{\mathtt{LO}} < \alpha \leq T_k$. In summary, the maximum net increase in workload is upper bounded by $(C_k^{\mathtt{HI}} - C_k^{\mathtt{LO}})$ for any left shift of the window in the reference pattern.

Rightward Shift: The workload in the shifted window due to right shift of the problem window can increase only if the window is shifted right for more than $(T_k - C_k^{\mathtt{HI}})$ time units. This is because no job of task τ_k executes within $[r_i^x + t, r_i^x + t + (T_k - C_k^{\mathtt{HI}}]$ in the reference pattern.

If $(T_k - C_k^{\mathtt{HI}}) < \alpha \leq (T_k - C_k^{\mathtt{LO}})$, then shifting the problem window right by α time units, the workload in the shifted window increases by $(\alpha - (T_k - C_k^{\mathtt{HI}}))$ time units. Since $\alpha \leq (T_k - C_k^{\mathtt{LO}})$, the maximum ***net*** increase in workload is $(C_k^{\mathtt{HI}} - C_k^{\mathtt{LO}})$, whenever $(T_k - C_k^{\mathtt{HI}}) < \alpha \leq (T_k - C_k^{\mathtt{LO}})$.

Every right shift of the problem window for exactly T_k time units must decrease the workload from the left-hand side by C_k^{LO} time units. Therefore, the workload within the shifted window is decreased by at least $(\alpha - (T_k - C_k^{\mathsf{LO}}))$ time units from the left-hand side for any right shift of the problem window by α time units whenever $(T_k - C_k^{\mathsf{LO}}) \leq \alpha \leq T_k$. Any right shift of the problem window by α time units, where $(T_k - C_k^{\mathsf{LO}}) \leq \alpha \leq T_k$, increases the workload within the shifted window by at most $(\alpha - (T_k - C_k^{\mathsf{HI}}))$ time units. Consequently, the maximum ***net*** increase in workload within the shifted window is equal to $(C_k^{\mathsf{HI}} - C_k^{\mathsf{LO}})$, whenever $(T_k - C_k^{\mathsf{LO}}) \leq \alpha \leq T_k$. In summary, the maximum net increase in workload is upper bounded by $(C_k^{\mathsf{HI}} - C_k^{\mathsf{LO}})$ for any right shift of the window in the reference pattern. □

B

Additional Graphs for Iterative Tests

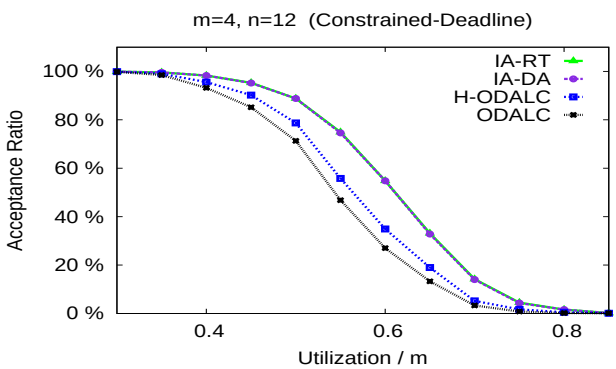

Figure B.1: *Acceptance ratios for experiments with* $(m = 4, n = 3m = 12)$.

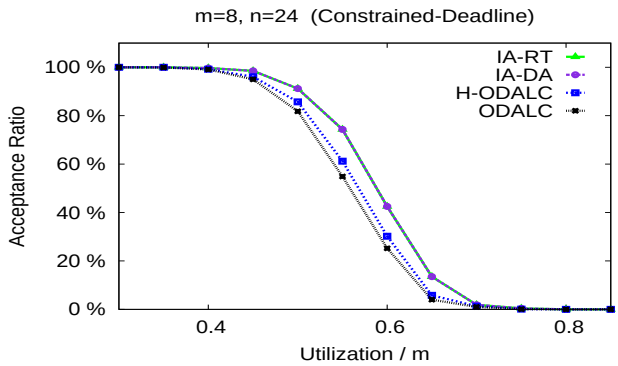

Figure B.2: *Acceptance ratios for experiments with* $(m = 8, n = 3m = 24)$.

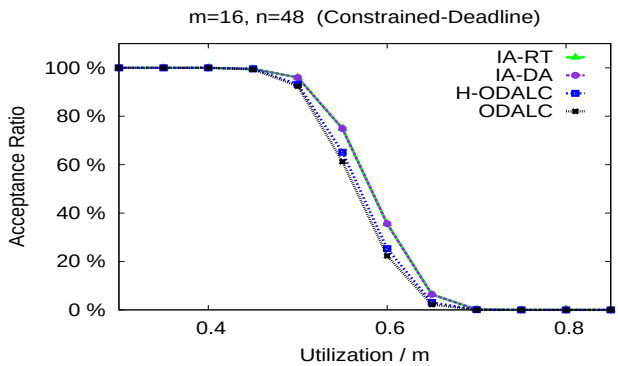

Figure B.3: *Acceptance ratios for experiments with* $(m = 16, n = 3m = 48)$.

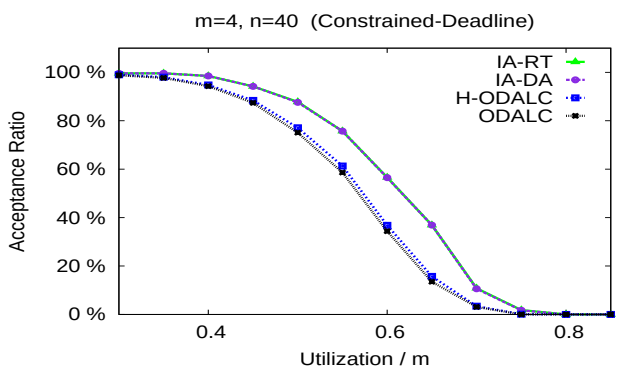

Figure B.4: *Acceptance ratios for experiments with* $(m = 4, n = 10m = 40)$.

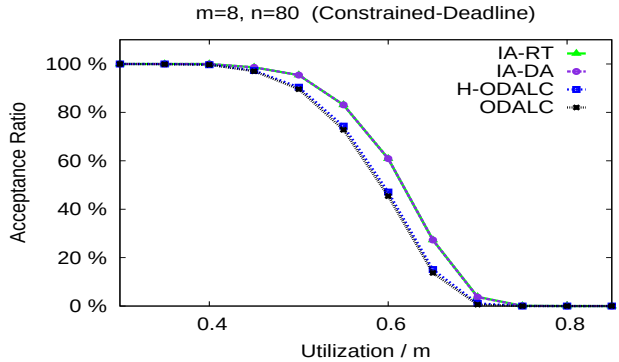

Figure B.5: *Acceptance ratios for experiments with* $(m = 8, n = 10m = 80)$.

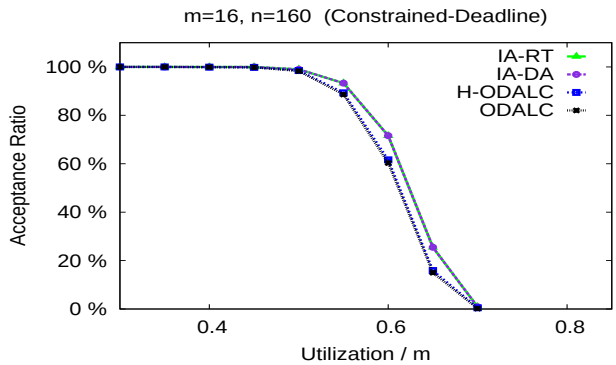

Figure B.6: *Acceptance ratios for experiments with* $(m = 16, n = 10m = 160)$. *The task set generation algorithm failed to generate 1000 task sets at utilization level beyond 70% for the given discard limit of 1000. So, the algorithm was aborted. However, the acceptance rationof all the tests are zero at utilization level* 70%.