REAL-TIME SCHEDULING OF MIXED-CRITICAL WORKLOADS UPON PLATFORMS WITH UNCERTAINTIES

Zhishan Guo

A dissertation submitted to the faculty at the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2016

Approved by:

Sanjoy K. Baruah

James H. Anderson

Alan Burns

Kevin Jeffay

Ketan Mayer-Patel

©2016
Zhishan Guo
ALL RIGHTS RESERVED

ABSTRACT

ZHISHAN GUO : Real-Time Scheduling of Mixed-Critical Workloads upon Platforms with Uncertainties
(Under the direction of Sanjoy K. Baruah)

In designing safety-critical real-time systems, there is an emerging trend in moving towards *mixed-criticality* (MC), where functionalities with different degrees of importance (i.e., *criticality*) are implemented upon a *shared* platform. Since 2007, there has been a large amount of research in MC scheduling, most of which considers the *Vestal Model*. In this model, all kinds of uncertainties in the system are characterized into the workloads by assuming multiple *worst-case execution time (WCET)* estimations for each execution (of a piece of code).

However, uncertainties of estimations may arise from different aspects (instead of WCET only), especially upon more widely used commercial-off-the-shelf (COTS) hardware that typically provides good average-case performance rather than worst-case guarantees. This dissertation addresses two questions fundamental to the modeling and analyzing of such MC real-time systems: (i) Can Vestal model be used to describe all kinds of uncertainties at no significant *analytical* capacity loss? (ii) If not, can new mechanisms be developed with better performances over existing ones (in MC scheduling theory), under certain assumptions?

To answer these questions, we first investigate the Vestal model carefully. We propose a new algorithm (named LE-EDF) which dominates state-of-the-art schedulers for MC job scheduling. We also improve the understanding of certain existing algorithms by proving a better (and even optimal) speedup bound. We have found that by introducing the probabilistic WCET workload model into MC scheduling, the uncertain behaviors can be better characterized comparing to Vestal model in the sense of *schedulability ratio* via experiments.

We then present a new MC system model to describe the uncertainties arising from the platform's performance. We show that under this model, where uncertainties of execution speed are separately captured, better schedulability results can be achieved compared to using the Vestal model instead. We propose a linear programming (LP) based algorithm for scheduling MC job set on uniprocessor platforms, and show its optimality (i.e., with zero *analytical* capacity loss), in the sense that it *dominates* any existing MC scheduler. Under the fluid (processor sharing) scheme, we further show that the optimality result can be retained even when the work is extended to multiprocessor scheduling and MC task scheduling.

This thesis further addresses the two questions by studying cases where uncertainties arise from more than one aspect, by integrating both dimensions of uncertainties (i.e., WCET estimation and system performance) within a single integrated framework and designing scheduling algorithms with associated schedulability tests. The proposed LE-EDF algorithm is shown to be well applicable for MC job scheduling. While For MC task scheduling, we adapt an existing algorithm named EDF-VD, and show that it has the same worst-case *analytical* capacity loss; i.e., the framework generalization is available "for free" at least from the perspective of speedup factor.

Under many cases, experimental studies upon randomly generated workloads are conducted to verify and quantify the theoretically proven *domination* relationships for both uniprocessor and multiprocessor scenarios.

Dedicated to my parents.

ACKNOWLEDGEMENTS

This dissertation and my Ph.D. degree would not have been possible without the help of many people.

First of all, I am deeply grateful to my advisor, Sanjoy Baruah, for his continued kindness, guidance, and support over the past four years. It has been such a great pleasure to work with and to learn from Sanjoy — his knowledgeableness, thoroughness, and non-aggressiveness (patience) have built a profound impression on how mentorship and research should ideally be conducted. I greatly appreciate Sanjoy for leaving me the space and freedom to think about and develop the research that I am interested in. Sanjoy is not only a wonderful academic advisor but also a nice friend and mentor in my daily life, who gently reminds me of things I can improve upon, and graciously offered help whenever I needed it. My life as a Ph.D. student has been so joyful and fruitful — Sanjoy is no doubt the key to that.

I am extremely thankful to James (Jim) Anderson, who has been a constant source of invaluable suggestions and encouragement in the real-time systems group. Although Jim is not my academic advisor, I have learned (and benefited) a lot from his passion and professionalism in both teaching and mentoring.

I would also like to express my sincerest appreciation to other members of my dissertation committee: Alan Burns (serving from across the Atlantic), Kevin Jeffay (serving as the department chair), and Ketan Mayer-Patel. Thank you for providing me the wise advice and insightful comments on my dissertation! Other faculty members in UNC who I owe gratitudes are: Wei Wang and Leonard McMillan for their support and advice in my first-year Ph.D. study; Frederick (Fred) Brooks, Gary Bishop, Ming Lin, and Jack Snoeyink (in addition to Sanjoy and Jim) for advising my teaching in Spring 2015 — my Departmental Teaching Award would not be possible without their assistance and guidance; and Don Smith and Shahriar Nirjon for their kind advice in my academic

job hunting. I owe my first academic position to the letters of recommendation written by Sanjoy, Jim, Fred, Gary, in addition to my M. Phil. advisor Jun Wang.

I am also grateful for the fruitful discussions and kind advice I received from my research co-authors or collaborators (in addition to my advisor): Wei Cheng, Arvind Easwaran, Nathan Fisher, Vladimir Jojic, Cong Liu, Eric Yi Liu, Luca Santinalli, and Kecheng Yang. I especially want to thank Wei who provided me many opportunities to work on interesting machine learning problems; Arvind who advised my job hunting and future research; Cong who invited me to UT-Dallas and gave unreserved suggestions during my job hunting; Luca who kindly hosted my memorable visit to ONERA; and Kecheng for the continued discussions and friendship. These talented people have enhanced my enthusiasm and understanding of computer science research, and it is my honor and pleasure to work with them. I am also indebted to Liliana Cucu-Grosjean and Robert Davis (other than Sanjoy) for inviting and supporting me to Dagstuhl Seminar 15121 — which further "embedded" me to the real-time systems community and greatly expanded my understandings to Mixed-Criticality.

It has been a great pleasure to spend five years in the Department of Computer Science at UNC. I want to thank all members in the real-time systems group, from whom I learn a lot — some members with whom I have interacted with the most are: Bipasa Chattopadhyay, Glenn Elliott, Namhoon Kim, Haohan Li, Cong Liu, Malcolm (Mac) Mollison, Bryan Ward, and Kecheng Yang. I am especially grateful to Haohan and Cong for introducing the wonderful group to me and guiding me through the beginning of my real-time research career; to Bipasa for sharing dozens of your carefully prepared hand-written notes; and to Mac and Bryan for giving valuable suggestions on many of my presentations including the job talk. I would like to take this opportunity to extend my appreciation to the administrative and technical staffs of the Computer Science Department. Special thanks to Jodie Gregoritsch (formerly Turnbull) for her extraordinary service and gracious help during my study. Life in Chapel Hill would not have been so enjoyable without my friends in the department and university, including (but are not limited to): Wei Cheng, Tianqu Cui, Yunchao

Gong, Yi Hong, Yuan Jin, Weibo Wang, Qianwen Yin, Zhaojun Zhang, and Jinsheng Zhou — thank you for sharing a great time with me in North Carolina!

Foremost, I thank my family for their unconditional love and inspiration during my Ph.D. study, and this dissertation is dedicated to them. My parents' successful and admirable lives as professors have always been a most important motivation for me to seek an academic career. Mother, thank you for being such a great educator in my early days. You built my self-confidence and talent in science. Thank you for always being supportive, understanding, and encouraging to me. Father, thank you for helping me develop the ambition in choosing CS as the major and lightening out my road ahead when I felt lost. Without my parents, this dissertation would not have been possible. Finally, I am deeply thankful to my wife Ning for all the love, trust, protection, and patience; for the kind support during the dark days of my life; for coming to the RTP area with me halfway around the world (although joining Duke, the rival of UNC); and especially for bringing Susan, our daughter, into our lives — ever since then I have been so energetic and happy. I could not have finished this dissertation without you.

TABLE OF CONTENTS

LIST OF TABLES xiv

LIST OF FIGURES xv

LIST OF ABBREVIATIONS xvii

1 Introduction 1

1.1 Motivation 2

1.1.1 Limitation of Traditional Models 3

1.1.2 Mixed-Criticality Design and Its Current Narrative 4

1.2 Thesis Statement 5

1.3 Contributions and Organization 6

2 Background 8

2.1 Workload Model 9

2.1.1 The WCET Abstraction 9

2.1.2 One-Shot Job Model 10

2.1.3 Sporadic Task Model 11

2.2 Computing Platform Model 13

2.3 Schedulers and Schedulability Tests 14

2.3.1 Classification of Schedulers 15

2.3.2 Common Uniprocessor Schedulers 16

2.3.3 Multiprocessor Schedulers 17

2.3.4 Optimality and Speedup 18

2.3.5 Schedulability Test 19

2.4 Mixed-Criticality Systems 20
2.4.1 MC Correctness and Modes 20
2.4.2 Vestal's Interpretation 21
2.4.3 Related MC Schedulers 23
3 When MC Arises from WCET Estimations 26
3.1 MC Job Scheduling on a Uni-Processor 27
3.1.1 System Model 27
3.1.2 Algorithm LE-EDF 29
3.1.2.1 Sub-Job Construction (LE) 29
3.1.2.2 Run-Time Scheduling (EDF) 33
3.1.3 Comparison over OCBP 35
3.1.4 Comparison over MCEDF 38
3.1.5 Experimental Comparisons 39
3.1.5.1 MC Job Generator 39
3.1.5.2 Schedulability Comparison 42
3.2 MC Task Scheduling on Uni-Processor 44
3.2.1 Motivation and Prior Work 45
3.2.2 Model 49
3.2.3 Probabilistic Schedulability 51
3.2.3.1 On the WCET Dependencies 53
3.2.3.2 Utilization Costs 54
3.2.4 Scheduling Strategy 56
3.2.4.1 The LFF-Clustering Algorithm 56
3.2.4.2 The Convolution Based Approach 60
3.2.4.3 Schedulability Test 61
3.2.5 Schedulability Experiments 67

3.2.5.1 MC Task Generator 67
3.2.5.2 Schedulability Comparison 68
3.3 MC Task Scheduling on Multi-Processor 71
3.3.1 System Model and Prior Work 71
3.3.2 Algorithm MCF 72
3.3.3 Correctness of MCF 76
3.3.4 Speedup of MCF and MC-Fluid 79
3.4 Summary 85
4 When MC Arises from Varying-Speed Platforms 87
4.1 Our MC Interpretation: The Varying-Speed Platform MC Model 87
4.2 MC Job Scheduling on Self-Monitored Uniprocessor 90
4.2.1 Model and Relationship to Prior Work 90
4.2.2 Algorithm TDMC-LP 92
4.2.3 Properties of TDMC-LP 101
4.2.4 The Dual-Criticality Sub-Case 103
4.2.4.1 A More Efficient Algorithm 103
4.2.4.2 An Optimization Version 107
4.2.5 NP-Hardness for Non-Preemptive Scheduling 111
4.3 MC Job Scheduling on Non-Monitored Uniprocessor 115
4.3.1 Motivation 115
4.3.2 An OCBP Based Algorithm 116
4.3.3 An Optimization Version of the Problem 120
4.3.4 Quantifying the Benefits of Self-Monitoring 121
4.3.5 The Speedup Cost of Not Monitoring 123
4.4 MC Job Scheduling on Multiprocessor 130
4.4.1 Model and Preliminary Results 130

4.4.2 Step 1 — A Linear Program ... 132
4.4.3 Optimal Run-Time Strategy for Degraded Mode ... 136
4.4.4 Optimal Run-Time Strategy for Weakly Degraded Mode ... 141
4.4.5 Necessity of Processor-Sharing ... 146
4.5 MC Task Scheduling on Uniprocessor ... 148
4.6 Summary ... 150
5 When MC Arises from More Than One Dimension of Uncertainties ... 152
5.1 Scheduling MC Job Set upon Varying-Speed Platforms ... 152
5.1.1 LE-EDF' — Enhanced LE-EDF ... 154
5.1.2 Online Optimality Under Single WCET Case ... 160
5.1.3 The Speedup of Non-Clairvoyance ... 162
5.2 Scheduling MC Task Set upon Varying-Speed Platforms ... 165
5.2.1 Model and Definitions ... 165
5.2.2 Non-Monitoring Processors ... 167
5.2.3 Self-Monitoring Processors ... 170
5.2.4 Experimental Evaluation ... 174
5.3 Summary ... 175
6 Conclusion ... 177
6.1 Summary of Results ... 177
6.2 Other Contributions ... 179
6.2.1 A Comparison of MC Job Models ... 179
6.2.2 Another Extension of the Vestal Model ... 180
6.2.3 A CPS Case Study on EDF Schedulability of AVR tasks ... 181
6.2.4 Solving MC Scheduling via a Neurodynamic Approach ... 182
6.2.5 Other Publications During Ph.D. Study ... 183
6.3 Future Directions ... 183

BIBLIOGRAPHY 185

LIST OF TABLES

2.1 Safety levels defined by different standards. 20

3.1 An example MC collection of jobs. 30

3.2 HI-criticality sub-jobs generated by Step 3 of LE-EDF in Example 3.2. 32

3.3 A set of MC tasks. 65

3.4 Example task system 74

4.1 An example of MC job set that is feasible upon a self-monitoring processor, while not schedulable upon an unmonitored one. 122

4.2 An MC job set that demonstrates the tightness of the speedup bound ϕ shown in (4.15). 129

LIST OF FIGURES

2.1 An execution pattern example of one-shot job set. 11

2.2 A release pattern example of a constrained-deadline sporadic task. 12

2.3 Varying execution lengths of the same task under different platform speeds............ 14

2.4 Classification of scheduling algorithms. 15

3.1 Schedulability comparison of OCBP (upper-left), MCEDF (lower-left), and LE-EDF (right, duplicated for easy comparison), where the color of each block represents the fraction of schedulable instances with ℓ_{LO} and ℓ_{HI} parameters falling within certain small ranges. (Informally, red is better – observe that there is almost no blue segment for LE-EDF – please view upon a color monitor/printout) 43

3.2 Schedulability comparison of EDF-VD (upper-left), pMC when returning partial correct or correct (lower-left), and pMC returning only correct (right, duplicated for easy comparison), where the color of each block represents the percentage of schedulable sets within certain *utilization* ranges. (*Please enlarge these figures enough, and/or use colored printer for better view.*) 69

3.3 Schedulability ratio comparison of EDF-VD and pMC, where HI utilization varies from 0.9 to 1 in a uniform manner.. 70

3.4 The run-time scheduling strategy used by Algorithm MC-Fluid........................ 73

3.5 Algorithm MCF 74

3.6 Plot of $f(x)$ for $c = 1$ (made with the WolframAlpha® computational knowledge engine: `https://www.wolframalpha.com/`).................................. 80

4.1 Linear program for constructing the scheduling table................................. 96

4.2 Basic steps of the proposed scheduling algorithm TDMC-LP. 97

4.3 The MC job set considered in Example 4.4, with the graphical depictions. 98

4.4 The constructed scheduling table of Example 4.4.....................................100

4.5 All MC jobs considered in Example 4.7, where a_i, c_i, and d_i stands for release date, WCET, and deadline respectively. 105

4.6 Degraded speed as a function of HI-criticality load and total load, where *ave* stands for average value and *std* stands for standard deviation 110

4.7 Degraded speed as a function of HI-criticality load, where *ave* stands for average value and *std* stands for standard deviation ... 111

4.8 Mixed-criticality instance considered in Example 4.10. ... 118

4.9 Determining the smallest degraded processor speed. ... 121

4.10 Lower bound on the speedup factor ϕ as a function of s - the degraded processor speed. 128

4.11 Linear program for determining the amounts to be finished for each job within each interval. ... 135

4.12 The schedule constructed by Wrap-Around-MC under normal mode in Example 4.22. ... 139

4.13 The schedule constructed by Wrap-Around-MC under a given degraded mode in Example 4.22. ... 140

4.14 The incorrect schedule constructed by Wrap-Around-MC (left), and a feasible one (right) under weak degraded mode in Example 4.24. ... 142

4.15 The schedule constructed by Level-MC under weak degraded mode in Example 4.25. .. 143

4.16 Joint execution of all jobs on the system by Level Algorithm during $[0.5, 0.9)$ of Example 4.25. ... 144

5.1 An example MC collection of jobs. ... 155

5.2 HI-criticality sub-jobs generated by Step 3 of LE-EDF' in Example 5.3. ... 157

5.3 VDF-NM: The preprocessing phase. ... 168

5.4 Example outcome of schedulability experiments, for parameters $[U_L, U_U] = [0.02, 0.2]; [T_L, T_U] = [5, 50]; [Z_L, Z_U] = [1, 4]; P = 0.5, s = 0.8$. The lowest line represents VDF-NM, the middle line represents VDF-NM+, and the top line represents VDF-WM. ... 175

LIST OF ABBREVIATIONS

WCET	Worst-Case Execution Time
EDF	Earliest Deadline First
FAA	Federal Aviation Administration
SWaP	Size, Weight, and Power
COTS	Commercial Off-The-Shelf
CPS	Cyber-Physical System
MC	Mixed-Criticality
CPU	Central Processing Unit
MPEG	Moving Picture Experts Group
CE	Cyclic Executive
RM	Rate Monotonic
SIL	Safety Integrity Level
FPH	Failure probability Per Hour
pWCET	probabilistic Worst-Case Execution Time
pET	probabilistic Execution Time
CDF	Cumulative Distribution Function
CCDF	Complementary Cumulative Distribution Function
LFF	Largest Fit First
LP	Linear Program
AVR	Adaptive Varying-Rate
ECU	Electric Control Unit
RNN	Recurrent Neural Network

CHAPTER 1: INTRODUCTION

Real-time systems are ubiquitous, ranging from portable devices like heart monitor watches and smartphones to large pieces of equipment such as nuclear power plant controllers and Mars exploration rovers. Computations in such systems need to be *logically* correct (as in general computing systems) and *temporally* correct; i.e., not only the result need to be mathematically sound, computations must also complete within their given time frames. The lack of temporal correctness in many real-time systems may lead to catastrophic results. For example, the response time of the throttle control in an avionic system must be small enough (e.g., within tens or hundreds of milliseconds) to guarantee that all required temporal constraints are satisfied at run-time in a predictable manner[1].

As a result, the temporal correctness of a real-time system needs to be demonstrated and verified prior to runtime; i.e., during the design and implementation process. Under any possible execution of the system, the designer must guarantee the temporal correctness of the computations. Unfortunately, it is too costly or even impossible to verify the temporal correctness of a hard real-time system via exhaustive simulation or testing, as the number of possible execution scenarios is prohibitively large even for very simple systems. Therefore, formal analysis techniques are necessary to ensure that the designed real-time systems are provably correct and predictable, which typically include three steps:

(i) Formally *modeling* the system;

[1] During typical landing processes, right after confirming the aircraft is on the ground, the throttle is set to full reverse to reduce the speed of the airplane at highest deceleration rate. Throttle levers are then set to idle when the plane is decelerating through a certain speed range since reverse thrust at a low speed will permanently damage the engines. In such safety-critical real-time systems, any failure of meeting the timing constraints may be crucial.

(ii) Choosing or designing a proper *scheduling strategy*; and

(iii) Deriving *schedulability tests* to validate temporal correctness at design time.

These three basic steps are fundamentally connected to each other. In short, every scheduling algorithm should have an associated schedulability test, and their analytical capacity losses are often greatly affected by how system behaviors are being modeled.

In this dissertation, we study how various kinds of models of uncertainties in mixed-criticality real-time systems affect the scheduling problem (in the sense of intractableness), and lead to strategies with different analytical capacity losses. This chapter gives a brief introduction to the whole document. The motivation will be described in the next section, followed by the thesis statement, and finally the contribution and organization.

1.1 Motivation

Safety critical systems, such as avionic and automotive systems, need to meet certain certification requirements before being qualified for implementation and application. For example, the Federal Aviation Administration (FAA) will verify the safety standards within a newly developed aircraft system, including the guaranteed temporal correctness of executions to the safety-critical functionalities. These authorities tend to be very conservative in the certification process, such that the correctness often needs to be demonstrated under extremely rigorous and pessimistic assumptions.

Certifications are based on the analysis of *models* of systems, rather than to the physical systems themselves. In order to have confidence that the conclusions drawn on the basis of the scheduling theories will hold for the actual systems (being modeled), the modeling process typically incorporates considerable pessimism. Such pessimism is unavoidable due to the uncertainties of system behaviors during run-time, such as WCET estimations and release patterns of workloads, as well as the run-time performances of the processor.

1.1.1 Limitation of Traditional Models

Traditionally, safety-critical real-time systems are rather simple and behave deterministically, in the sense that there are only basic functionalities to be implemented upon special-purpose hardware platforms that are often built to behave highly reliable. Due to such design-time predictability, the actual "cost" for making pessimistic assumptions remains low, such that classic real-time scheduling theory is developed upon very simple workload and platform *models*. We assume a single worst-case execution time (WCET) for each function and a constant speed for the platform. As the *modeling* step of formal real-time analysis is not a big issue, people focus more on answering the remaining two types of questions: (i) Which scheduling algorithm should one choose to run the given workload upon a computing platform? (ii) Can all timing constraints be met under a given scheduling strategy?

The limitation of choosing simple workload and platform models has become significant in the 21st century due to several facts. First of all, most chip manufacturers are shifting towards multi-core architectures to address the need to achieve higher performance without driving more power. Real-time applications often exhibit rather complex run-time behaviors on multi-core platforms as they often need to share memory and caches with other workload running in parallel. Also, due to size, weight, and power (SWaP) constraints, there is an emerging trend in building such systems on commercial-off-the-shelf (COTS) platforms, upon which various kinds of uncertainties arise, leading to a huge gap between the average-case and the worst-case execution behaviors. Finally, there is an emerging embedded system design trend towards building complex cyber-physical systems (CPS), e.g., self-driving vehicles, intelligent health-care devices, and smart power grids. Many computations on CPS interact with and depend upon the integrated physical elements, which often results in complex run-time behaviors. All these facts are leading to a tremendous growth in the gap between average-case and worst-case run-time behaviors for modern real-time systems. As the worst cases are highly unlikely to be revealed during actual runs, a huge portion of computational resource is being wasted during under the traditional over-provisioning design mechanism.

1.1.2 Mixed-Criticality Design and Its Current Narrative

Knowing the shortcomings of the traditional design, there is an emerging trend in the move towards *mixed-criticality* (MC) implementations of real-time systems, where functionalities with different degrees of importance are implemented upon a *shared* platform. Such an approach recognizes that the over-provisioned resource of the critical functionalities is highly unlikely to be used during run-time (due to very conservative assumptions made), and can be used to execute the less-critical functionalities instead. The routine has been to validate the correctness of highly critical functionalities under more pessimistic assumptions than the assumptions used in validating the correctness of less critical functionalities. All the functionalities are expected to be demonstrated correct under the normal analysis, whereas the analysis under the more pessimistic assumptions needs only demonstrate the correctness of the more critical functionalities.

Mixed-Criticality arises naturally in many real-time systems, with different numbers of criticality levels in different applications. For example, in the RTCA DO-178B avionics software standard, the tasks are classified into five assurance levels, from level A to level E. In the standard, a failure of a level-A task will have catastrophic results (e.g. causing a crash), while a failure of a level-E task will have no influence on flight safety.

Those MC real-time systems, like per-criticality-level isolated (i.e., single criticality) ones, need to pass safety certification as well, yet the deadlines of workloads with less importance may be missed *occasionally*. Such integration results in a risk of having non-critical components affecting the behavior of critical ones during run-time — new tools, techniques, and methodologies *must* be derived to prevent such failures. In 2007, Steve Vestal (Vestal, 2007) proposed a multi-WCET workload model and formally defined the correctness of an MC system as per-mode basis: actual executions of functionalities may trigger a mode switch to the whole system (as their executions exceed certain WCET thresholds), leading to correctness guarantees to different sets of workloads. Under such design, less important deadlines are guaranteed to be met when all executions signal their finishing upon less pessimistic WCET estimations[2].

[2]Please refer to Chapter 2 for the formal definition and detailed description of MC correctness.

Apparently, Vestal's attempt belongs to the modeling step among the aforementioned three steps in formal real-time analysis. A large amount of research has been done in the past 8 years on MC scheduling under the Vestal Model (see (Burns and Davis, 2016) for an up-to-date review, or Section 2.4.3 for the description of some related work). Unfortunately, most of the work only focuses on the latter two steps; i.e., developing and analyzing new schedulers for MC systems — the real-time system community rarely reviews the first step: modeling.

In this dissertation, we revisit the modeling step as well, and proposed new MC model based on the varying-speed platform. Uncertainties arise not only from WCET estimations, but also from estimations of platform's execution speeds. Conditions during run-time, such as changes in the ambient temperature, the supply voltage, etc., may result in variations in the *clock speed* — for instance, a system programmer may use the userspace Linux command `cpuspeed` to configure a system to reduce the clock speed of the central processing unit (CPU) if the core temperature becomes too high. At the hardware level, too, innovations in computer architecture for increasing clock frequency can lead to variable-speed clocks during run-time: e.g., (Bull et al., 2010) describes a novel technique for detecting whether signals are late at the circuit level within a CPU micro-architecture, and if so to recover by delaying the next clock tick so that logical faults do not propagate to higher (i.e., the software) levels.

Similar to the case for uncertainties in WCET estimation, uncertainties in processing speed may lead to significant under-utilization of the CPUs computing capacity: in order to guarantee temporal correctness to all functionalities under all circumstances, one must make the most pessimistic assumptions regarding clock speed: during run-time the clock speed takes on the lowest possible value, which could be highly unlikely to be reached in practice. A natural question arises, is Vestal model still representative enough to cover other kinds of uncertainties in MC real-time system?

1.2 Thesis Statement

As stated above, the modeling step is tightly related to the developing and analyzing schedulers for real-time systems. Thus, to answer the important question of whether we have paid enough

attention to the modeling step of analyzing MC real-time systems, one needs to examine cases where uncertainty of estimations arise from different aspects and compare schedulers and their associated tests with existing ones based on Vestal Model. This leads to my thesis statement as follows:

> *In extending mixed-criticality real-time system design and analysis to systems where uncertainty of estimations arise from different aspects, existing scheduling methods may be adapted at no significant capacity loss in some cases, while in some other cases new mechanisms can be developed, with better performance (over existing scheduling algorithms) shown theoretically, in the sense of proven domination relationship or better speedup bounds, and/or experimentally via simulations.*

1.3 Contributions and Organization

The above thesis is supported by the following contributions made in this dissertation:

- Chapter 2 presents the real-time workload and platform models considered in this dissertation, and provides necessary background information on real-time scheduling theory as well as MC systems.
- Chapter 3 studies the MC scheduling problem under Vestal Model, where uncertainties arise from the WCET estimations. We are able to improve the current state of the art by (i) deriving new scheduling algorithms that either dominate or outperform existing schedulers both theoretically and experimentally; (ii) adding a parameter to the MC workload model and deriving more efficient scheduling strategy under probabilistic analysis; and (iii) Mathematically proving better speedup result for existing algorithms.
- Chapter 4 proposes a new model dealing with uncertainties that arise from execution speed of the platform. New optimal scheduling strategies are identified with associated schedulability tests. Experimental comparisons against existing methods with Vestal Model suggest that

such kind of uncertainty is worth being separately modeled, at least from the scheduling theory point of view.

- Chapter 5 further integrated both dimensions of uncertainties within a generalized framework. For MC job scheduling, we show the proposed LE-EDF algorithm retains online optimal property, while for MC task scheduling, existing scheduler (EDF-VD) can be adapted with reasonable schedulability lost according to speedup bound analysis.

- Chapter 6 summarizes the work, lists some other contributions, and discusses about future research directions motivated by this dissertation.

CHAPTER 2: BACKGROUND

In real-time systems, one needs to ensure all timing constraints be met under a given scheduling algorithm. However, we rarely analyze an actual system directly — it is the *model* of the system that we are scheduling, which includes characteristics of the workload, the computational platform, the scheduling algorithm, etc. Good models characterize a system at the proper abstraction level, such that unnecessary (or non-relevant) details of system behaviors are blocked from the scheduler, while important information remain revealed, such as timing constraints for validating the temporal correctness of the system.

Other than the scheduler itself, there are two important elements in real-time systems: workloads, which are pieces of codes to be executed; and platforms, upon which the codes are being executed. This chapter mainly introduces some workload and platform models in real-time scheduling theory. Some common definitions, notations, and prior work will be provided as well.

Real-time systems are becoming more and more complicated in their workloads (advanced features or functionalities are being implemented) as well as their computational platform structures (as evidenced by, e.g., the shift to multi-core systems in early 2000's). As a result, the models (and associated schedulers) people use to study real-time schedulability is evolving as well. Various kinds of workload models have been proposed in the past few decades — one may find (Buttazzo et al., 2014) a useful resource for tracking other real-time system workload models.

Regarding prior work on schedulers, we will only give brief introductions to them in Sec. 2.4.3 — more detailed descriptions of some closely-related algorithms will be elucidated in each of the following chapters or sections separately. The main reason to organize the dissertation in such a way is that one scheduling algorithm may be used for various kinds of workload with different performances. As our attention may shift for each chapter, we believe it is reasonable to give a fresh

and focused review for related prior work within each chapter, and hope such organization makes each chapter more self-contained and the reading experience less boring.

2.1 Workload Model

Real-time workload models are designed to describe real-time applications and their associated temporal constraints mathematically. As discussed in the previous chapter, designers have to make conservative assumptions in the modeling process in order to provide guarantees to the temporal correctness of real-time systems. *Predictability* of the uncertainties is essential for real-time systems and is often achieved by necessary *a priori* knowledge of applications running in the system. Workload models reveal such knowledge.

This section describes two classic real-time workload models: the one-shot job model and the sporadic task model. Both models are based on one key concept: the WCET abstraction, which is introduced in the first subsection.

2.1.1 The WCET Abstraction

The WCET abstraction plays a central role in the analysis of real-time systems. For a specific piece of code and a particular platform upon which this code is to execute, the WCET of the code denotes an upper bound on the amount of time the code takes to execute upon the platform. Determining the exact WCET of an arbitrary piece of code is provably an undecidable problem. Devising analytical techniques for obtaining tight upper bounds on WCET is currently a very active area of research, and sophisticated tools incorporating the latest results of such research have been developed (see (Wilhelm et al., 2008) for an excellent survey).

As WCET tools are more or less conservative than each other, multiple WCET bounds can be provided for a single piece of code. It is often the case that different WCET values reflect different confidence or *certification levels*, and WCET bounds with higher confidence may be achieved by multiplying (the provided WCET) by a fudge factor which is greater than 1.

2.1.2 One-Shot Job Model

The basic unit of computation in real-time scheduling is called a *job*. A job is an abstraction of one execution of a piece of work. Some codes are executed repetitively in a system — those are described in the following section.

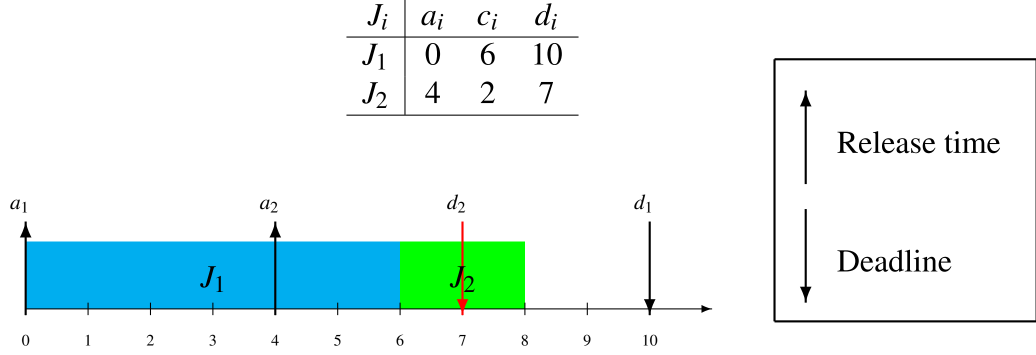
We denote a real-time job as J_i, where i is its identity index. A job can be characterized by a 3-tuple of parameters: $\{a_i, c_i, d_i\}$, where

- $a_i \geq 0$ denotes its release time (the first moment that the piece of code can start to execute),
- $c_i \in \mathbb{R}_+$ is the WCET estimation, and
- $d_i \geq a_i$ indicates the deadline (upon which the job should be finished).

From the system scheduler's point of view, a job becomes ready to execute only when it signals its arrival. It is the scheduler's responsibility to guarantee its temporal *correctness*; i.e., to receive up to c_i time units of execution within the interval $[a_i, d_i)$, known as its *scheduling window*.

The scheduler makes the decision of when to allocate a processor (or any computing unit) to the job, upon which the job starts to execute. The job will signal its completion when it finishes its execution. It is assumed that a job J_i may receive as long as c_i time units to complete its execution, but should not exceed that. Techniques like watchdog timer (Stajano and Anderson, 2000) can be used to suspend or terminate a job's execution when necessary, in order to guarantee c_i being an absolute upper bound.

Figure 2.1 shows one possible schedule of a set of two jobs, where J_1 is scheduled correctly while J_2 is not, with a missed deadline shown in red. We use up-arrows to denote release times, and down-arrows for deadlines for all figures of execution patterns in this dissertation. Different colors will be used for executions of different jobs, and thus, it is recommended that the reader views these figures upon a color monitor/printout.



J_i	a_i	c_i	d_i
J_1	0	6	10
J_2	4	2	7

Figure 2.1: An execution pattern example of one-shot job set.

2.1.3 Sporadic Task Model

Many works in real-time systems are being executed repetitively, such as decoding a MPEG (Moving Picture Experts Group) video frame in multi-media applications, or converting an analog sensor signal into a digital one in an avionic control software. The *implicit-deadline sporadic task* (Liu and Layland, 1973), also known as *Liu and Layland task*, or *task* in short, describes such kind of workload.

An *implicit-deadline sporadic task* τ_i is characterized by two parameters: its WCET C_i and a minimum inter-arrival separation T_i (also know as its *period*). Such a task may potentially generate an unbounded number of jobs, with the first among the series arrive at any time and subsequent releases being at least T_i time units apart. Each job has an execution requirement of as much as C_i time units, and its deadline is T_i time units after its release.

A relative deadline D_i (which is no greater than T_i) can be specified under the *constrained-deadline sporadic task* model (Mok, 1983) with three parameters: $\{C_i, D_i, T_i\}(D_i \leq T_i)$. To guarantee correctness, each job should receive enough execution by D_i time units after its arrival (which is more "constrained" than the implicit-deadline case)[1].

Figure 2.2 shows one possible release pattern of a constrained-deadline sporadic task $\tau_1 = \{1.5, 2, 3\}$, where the releases of the first two jobs ($\tau_{1,1}$ and $\tau_{1,2}$) are exactly $T_i = 3$ time units apart, while the third job $\tau_{1,3}$ does not arrive until $t = 8$, although it is "legal" for it to be released at $t = 6$

[1]We do not consider the *arbitrary* deadline task set, where D_i can be greater than T_i.

(indicated by the dashed arrow). Note that although there is flexibility in the release time of each job, the deadline always comes $D_i = 2$ time units after its arrival.

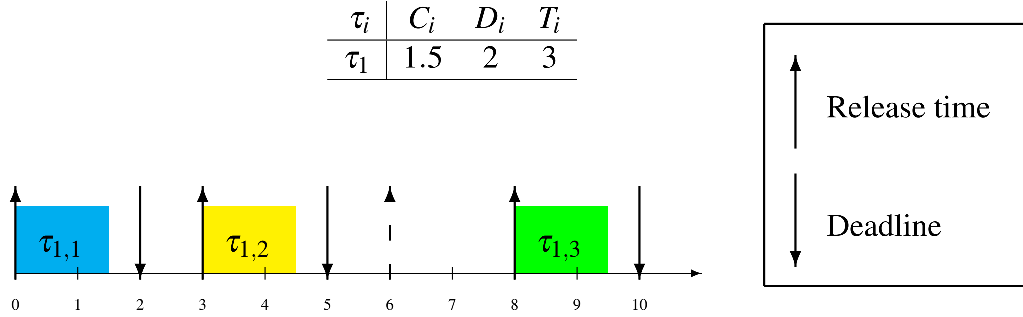


Figure 2.2: A release pattern example of a constrained-deadline sporadic task.

Two widely used concepts for sporadic tasks are *utilization* and *density*. *Utilization* of a task is defined as the ratio of the WCET parameter to its period; i.e.,

$$u_i = \frac{c_i}{T_i}. \tag{2.1}$$

Density is defined as the ratio of the WCET parameter of a task to its relative deadline; i.e.,

$$\delta_i = \frac{c_i}{D_i}. \tag{2.2}$$

If a *task set* τ contains n number of tasks $\tau_1, \ldots, \tau_n$, its total utilization is defined as the sum of the utilizations of each task; i.e.,

$$U = \sum_{i=1}^{n} u_i = \sum_{i=1}^{n} \frac{c_i}{T_i}. \tag{2.3}$$

Note that the sporadic task model can be specified into more basic workload models, such as the *periodic task* model. The release pattern of consecutive jobs of a periodic task τ_i is fixed as T_i time units apart.

2.2 Computing Platform Model

The computing platform is always an essential part for modeling a real-time system. Any WCET estimation is associated with a certain platform, i.e., the WCET of a given piece of code may vary dramatically upon different platforms.

It is commonly assumed in real-time systems research that a processor runs at a fixed speed of 1. Such assumption simplifies the execution pattern of each job, where its WCET parameter is reflecting the number of time units it may take (in the worst case) to finish a job.

However, as discussed in Section 1.1.2, conditions during run-time, such as changes in the ambient temperature, the supply voltage, etc., may lead to variations in the clock speed. A WCET tool must make the most pessimistic assumptions regarding the clock speed, leading to a significant under-utilization of the CPU's computing capacity during run-time. As a result, we adopt a more generalized computing platform model in this dissertation. The WCET tool still makes the assumption that clock speed remains at 1 for its estimation, but the actual run-time length depends (and could be larger than the WCET). It is assumed that a processor's main frequency may vary during run-time, and a job executing on a processor of speed s for t time units completes $s \times t$ units of execution.

Figure 2.3 shows the execution pattern of a periodic task $\tau_1 = \{2,3\}$, where the processor initially runs at the speed of 1 and suffers from a performance degradation (to speed 0.5) at time $t = 5$. The height of jobs indicates the execution speed at the moment. Under this case, the first job $\tau_{1,1}$ finishes its execution within 2 time units, while the second one takes 3 (which is longer than its WCET, $C_1 = 2$), and the third one takes 4 (which results in a deadline miss at $t = 11$, denoted in red in the figure).

A *multiprocessor* is a combination of multiple uniprocessors, and can be classified into one of the following platform models depending upon the relationship between the computing capacities of those processors:

- Identical: all processors run at the same speed, which is usually normalized to 1.

τ_i	C_i	D_i	T_i
τ_1	2	3	4

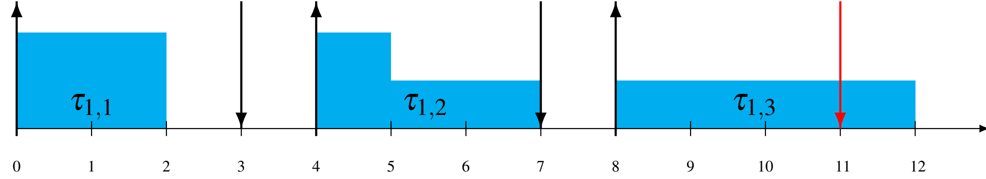


Figure 2.3: Varying execution lengths of the same task under different platform speeds.

- Uniform: each processor is characterized by its own execution speed, which may be different from the speed of other processors on the same platform.
- Unrelated: the length of an execution depends on both the processor and the task itself; i.e., a given processor may execute different tasks at different speeds.

In this dissertation, three types of (models of) platforms will be considered: uniprocessor, identical multiprocessor, and uniform multiprocessors.

2.3 Schedulers and Schedulability Tests

In general, given a set of tasks or jobs and a platform, we want to know what scheduling strategy guarantees its correctness. From the scheduling theory point of view, the goal becomes twofold: (i) proposing *good* scheduling algorithms (schedulers), and (ii) deriving the *schedulability tests*, which are the conditions to be satisfied in order to guarantee correctness of functionalities under a given strategy.

The analytical schedulability cost comes in twofold as schedulers and schedulability tests are tightly connected to each other. On one hand, it is the scheduler that decides which job(s) to be executed on which processor at any time instant, and a good decision may not be always easy to achieve. On the other hand, the best decision may result in complicated schedulers, where it becomes computationally infeasible to derive a *straightforward* schedulability test.

In this section, we first introduce the general classifications of scheduling algorithms, and then highlight the typical schedulers for each category. Schedulability tests and some useful definition will be described in the final part.

2.3.1 Classification of Schedulers

The broad classification of scheduling algorithms is based on when the schedule is generated. Under *static* (or *offline*) scheduling, the schedule is generated prior to run-time, which often requires the precise knowledge of arrival time and deadlines of all jobs. While under *dynamic* (or *online*) scheduling, the scheduling decision of a job is made *after* the job has arrives.

As shown in Figure 2.4, online schedulers can be further categorized into *fixed-priority* ones and *dynamic-priority* ones. Note that here the term "dynamic" refers to the priority assignment, *not* the time when scheduling decision is made. Both *fixed-* and *dynamic- priority* schedulers are dynamic scheduling strategies that make scheduling decisions during run-time, where the temporal behaviors of all released jobs need to be taken into account in order to determine which job should be prioritized over others.

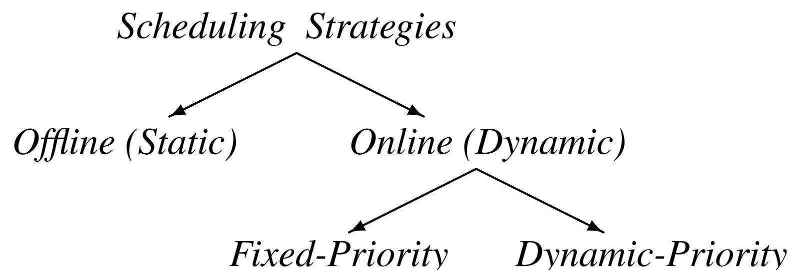


Figure 2.4: Classification of scheduling algorithms.

Under fixed-priority, priorities are assigned to tasks that all jobs of task τ_i will be prioritized over any job of another lower-prioritized task. In contrast, under dynamic priority, those jobs may have different priorities depending on their arrival time and the absolute deadlines. There are pros and cons for both dynamic- and static- priority schedulers. Dynamic-priority scheduling provides more flexibility, and thus can often better utilize the computing capacity of a resource.

However, study has shown that dynamic-priority often results in larger overheads as the number of preemptions and migrations tend to be larger. Also, under fixed-priority scheduling, a deadline miss of a task with priority level L can only be caused by tasks with higher priority, leading to much easier maintaining and debugging; while such nice properties are not shared by dynamic-priority systems.

2.3.2 Common Uniprocessor Schedulers

Cyclic Executive (CE) is an important way to sequence tasks or jobs offline in a real-time system. It is also known as table-driven scheduling, since a "look up table" Γ is calculated prior to run-time, defined as follows:

$$\Gamma(t_k) = \begin{cases} J_i, & \text{if } J_i \text{ is to be starting its execution at time } t_k; \\ I, & \text{if no task to be scheduled at time } t_k. \end{cases} \tag{2.4}$$

It is obvious that such table reveals all scheduling decisions during run-time. For example, the job set J mentioned in Figure 2.1 can be correctly scheduled under the table $\{(0,J_1),(5,J_2),(7,J_1),(8,I)\}$. Under such schedule[2], all deadlines could be met, whereas the second job fails to meet its deadline in the original schedule shown in Figure 2.1.

Such table driven manner makes the system very predictable while being extremely efficient in task dispatch, and thus dominates safety-critical systems historically. People have realized the significant drawbacks, such as (i) they are very brittle that any change of the set requires a new table to be computed, (ii) it is required that release patterns and deadlines must be *a priori* known, and (iii) the frame size could be huge, etc.

Rate Monotonic (RM) is a well-known fixed-priority algorithm, which assigns priorities to tasks based upon their periods: a shorter period leads to a higher priority. Under such a mechanism, a job released by a higher priority task will preempt any job with a lower priority.

[2]Note that job J_1 is "suspended" by J_2 at time $t = 5$, although J_1 remains unfinished. Such behavior is called a "preemption", which is allowed and assumed at zero cost in this dissertation.

Earliest Deadline First (EDF) scheduling maintains a priority queue of jobs, ordered by shortest time remaining to absolute deadlines. During run-time, the job at the front of the queue is chosen for execution and removed from the priority queue. At each time a new job is released, the queue will be updated, and a job that is *preempted* will be re-queued. Again, *preemption* is allowed and assumed at zero cost in this dissertation.

Virtual Deadline is a scheduling technique to provide more flexibility in decisions with deadline based schedulers like EDF. The virtual deadlines are provided for assigning priority and making scheduling decisions only. They may be different from actual deadlines to either increase or decrease the priority of a job (or task) dynamically.

2.3.3 Multiprocessor Schedulers

Embedded systems, especially safety-critical ones are increasingly implemented on multicore platforms. On such platforms, there are three main scheduling approaches: partitioned, global, and clustered scheduling.

Partitioned scheduling statically assigns each task to a dedicated processor. The processor assignment is fixed for all jobs released by the same task. The good thing about partitioned scheduling is that, once such assignment is provided, the multiprocessor scheduling problem becomes multiple uniprocessor scheduling problems that have been well studied. However, the partitioning step itself is computationally intractable — optimally assigning all tasks on processors is NP-hard in the strong sense (reduction from the Bin-Packing problem (Kellerer et al., 2004))

Global scheduling allows a job of a task to execute on any processor. A job may also migrate before it completes execution and executes on a different processor, which is called intra-job migration.

Fluid scheduling (Holman and Anderson, 2005) allows more than one job to be "executed" on a processing core *simultaneously*; i.e., each job can be regarded as executing on a dedicated "fractional" processor with executing speed no greater than 1. The execution rate of a task τ_i matches the definition of processor speed in Sec. 2.2. Fluid scheduling can provide the ideal allocation

of computing resources, but creates an unbounded number of preemptions which is not practical. Fortunately, techniques (e.g., (Baruah et al., 1996), (Cho et al., 2006), (Funk et al., 2011)) have been introduced to equivalently transform a fluid schedule into a unit-based non-fluid one with limited number of preemptions and migrations, which makes it applicable to real hardware platforms.

2.3.4 Optimality and Speedup

A *feasible* schedule indicates that every job completes by its deadline. A set of tasks is *schedulable* according to a scheduling algorithm if the scheduler always produces a feasible schedule.

An *optimal* scheduler would guarantee schedulability of *any* feasible task/job set; i.e., whenever a feasible schedule exists, it is schedulable by the optimal scheduler. For example, the following theorem suggests that EDF (described in Sec. 2.3.2) is optimal for uniprocessor job set and sporadic task set scheduling.

Theorem 2.1. *(Liu and Layland, 1973) If a real-time job set (or sporadic task set) is schedulable on a uniprocessor by any scheduling policy, it is also schedulable by EDF.*

As optimal is hard to achieve for many other cases, e.g., for multiprocessor and/or mixed-criticality we introduce a commonly used metric for comparing schedulability tests: *speedup* (Kalyanasundaram and Pruhs, 2000).

Definition 2.2. *A* speedup factor *of* $s (s \geq 1)$ *for a scheduler* $\mathscr{S}$ *implies that any task set that is schedulable on a platform of speed-*1 *processor(s) will be deemed schedulable by* $\mathscr{S}$ *on a platform with speed(s) increased to* s*.*

In short, speedup measures how "far away" is a given scheduler from an optimal one — it reflects the effectiveness of a scheduling policy. It is obvious that a speedup factor of 1 indicates optimal. While if optimal cannot be achieved (which is often due to computational intractability), we would want to propose schedulers and schedulability tests with smaller speedup (i.e., closer-to-1). Speedup factor will be used to measure many schedulers in this dissertation.

2.3.5 Schedulability Test

Given a real-time scheduling algorithm and a task set τ, we should be able to determine whether it is *schedulable* prior to run-time, which is the general purpose of a *schedulability test*. A schedulability test (which is often in the form of computational conditions) indicates whether all deadlines of a given task set can be met (i.e., correctness) under a specific scheduling algorithm. Schedulability tests should be computationally tractable, but not necessarily efficient as they are performed during the system design and analysis process instead of run-time.

Necessary schedulability test only provides necessary conditions for a task set to be schedulable by its associated algorithm; i.e., a task set will not be schedulable if it fails to satisfy any *necessary schedulability test*. Passing a *sufficient schedulability test* guarantees the correctness of a task set over certain strategy. Failing to pass a necessary schedulability test mean the set is not schedulable by the associated algorithm; while failing to pass a sufficient one means it may or may not be schedulable (one needs a better test).

Exact schedulability test is both necessary and sufficient. The following theorem presents such kind of a test for EDF schedulability to implicit-deadline sporadic task set on a uniprocessor platform.

Theorem 2.3. *(Liu and Layland, 1973) A real-time implicit-deadline sporadic task set is schedulable by EDF on a uniprocessor platform* if and only if *its utilization (defined in* (2.3)*) is no greater than* 1.

From Theorem 2.3, we can easily derive the following schedulability test for constrained-deadline sporadic task set τ, by transforming each task $\tau_i = \{C_i, D_i, T_i\}$ into an implicit-deadline sporadic task $\tau_i' = C_i, D_i$, and perform the same test. Note that Corollary 2.4 provides a sufficient schedulability test only, as such transformation includes pessimism — the newly constructed tasks (for easy analysis purposes) may release consecutive jobs in a period of D_i, which is shorter than the actual allowed minimum separation under original description (T_i).

Corollary 2.4. *A real-time constrained-deadline sporadic task set is schedulable by EDF on a uniprocessor platform* if *its density (defined in* (2.2)*) is no greater than* 1.

2.4 Mixed-Criticality Systems

Mixed-Criticality arises naturally in many real-time systems, with a different number of criticality levels in different applications. In the functional safety standard for both automotive systems (i.e., ISO 26262) and railway systems (i.e., CENELEC 50126/128/129), four Safety Integrity Levels (SILs) are defined. Table 2.1 illustrate the relationship between different criticality levels in typical system standards.

<table>
<tr><th>Standards</th><th>ASIL, ISO 26262</th><th>Class, IEC 62304</th><th>SIL, IEC 61508</th><th>SSIL, EN 50128</th></tr>
<tr><td rowspan="6">Safety Levels</td><td>-</td><td>-</td><td>-</td><td>0</td></tr>
<tr><td>A</td><td>A</td><td>1</td><td>1</td></tr>
<tr><td>B</td><td>-</td><td rowspan="2">2</td><td rowspan="2">2</td></tr>
<tr><td>C</td><td>B</td></tr>
<tr><td>D</td><td>C</td><td>3</td><td>3</td></tr>
<tr><td>-</td><td>-</td><td>4</td><td>4</td></tr>
</table>

Table 2.1: Safety levels defined by different standards.

In an MC system, it is assumed that each job or task is assigned a criticality level $\chi_i \in \{1, 2, ..., L\}$ (where $L \in \mathbb{Z}_+$ is the total number of criticality levels), expressing its degree of importance, with a larger value denoting a greater level of importance. For the dual-criticality special case, the two criticality levels are commonly denoted as LO and HI instead of 1 and 2.

2.4.1 MC Correctness and Modes

Those MC real-time systems, like per-criticality-level isolated (i.e., single criticality) ones, need to pass safety certification as well, though the less important deadlines may be missed occasionally. To formally define the *correctness* of such systems, Vestal (Vestal, 2007) proposed an MC model where a set of *system behaviors* are identified and linked to system execution *modes*. Such system behaviors may include a certain job exceeding its less pessimistic WCET estimation, or the speed

of a computing platform dropping below a certain threshold, etc. During run-time, different levels of correctness are guaranteed under different execution mode, though the precise manners such as mode switch time are not *a priori* known.

Definition 2.5. *An MC system is scheduled* correctly *if workloads with criticality level l meet their deadlines whenever the system is experiencing level-$\mathscr{L}$ behaviors, for any* $1 \leq l \leq L$*, and* $\mathscr{L} \leq l$*.*

Under this definition, the assumptions we made to the system uncertainties during the modeling process are scaled to L modes, and the correctness of the whole MC system includes separate correctnesses under each of the L modes; i.e., only deadlines with criticality level no lower than l are guaranteed to be met when the system exhibits level l behavior. It is the system designer and analyzer's responsibility to define the relationship between system behaviors and executing modes; e.g., when to trigger a mode switch.

2.4.2 Vestal's Interpretation

Under Vestal model (Vestal, 2007), the mode switch is solely triggered by executions of workload exceeding a certain threshold. Please note that the MC system model is not restricted to that, although Vestal's interpretation receives most attentions in existing work. The Vestal model is based on the fact that WCET tools are more or less conservative than each other, providing multiple WCET bounds (Wilhelm et al., 2008). In some cases, WCET bounds with higher confidence may also be achieved by multiplying a *fudge factor* (which is greater than 1). In general, when considering a piece of code with criticality level l, up to l WCET bounds are provided, forming a *WCET vector* with non-decreasing elements: $c_i = \{c_i^1, c_i^2, ..., c_i^l, ..., c_i^L\}$, where $c_i^l = c_i^{l+1} = ... = c_i^L$.
MC Job Model. An independent *job* characterizes a single piece of code, to be executed once upon a real-time platform. An *MC job* J_i can be represented by a 4-tuple of parameters (a_i, c_i, d_i, χ_i), where

- $a_i \geq 0$ denotes its release time (after which the piece of code can start to execute),
- $c_i \in \mathbb{R}_+^L$ is the WCET vector, with L non-decreasing elements

- $d_i \geq a_i$ indicates the deadline (upon which the job should be finished), and

- χ_i represents the criticality level.

In classic real-time scheduling theory (see, e.g.,(Liu, 2000, page 81)), the *load* of an instance of jobs denotes the maximum cumulative execution requirement by jobs of the instance over the interval, normalized by the interval length, over all time intervals. Informally, the load of an instance can be thought of as representing a lower bound on the speed of any processor upon which the instance can meet all deadlines. Analogous to this concept, we find it convenient to define two loads, $\ell_{\text{LO}}(J)$ and $\ell_{\text{HI}}(J)$, for any MC collection J of jobs.

Definition 2.6. *The* LO*-criticality load* $\ell_{\text{LO}}(J)$ *and the* HI*-criticality load* $\ell_{\text{HI}}(J)$ *of an MC collection* J *of jobs are defined according to the following two formulas:*

$$\ell_{\text{LO}}(J) = \max_{0 \leq t_1 < t_2} \frac{\sum_{J_i:\ t_1 \leq a_i \wedge d_i \leq t_2} c_i^L}{t_2 - t_1}; \tag{2.5}$$

$$\ell_{\text{HI}}(J) = \max_{0 \leq t_1 < t_2} \frac{\sum_{J_i:\ \chi_i = \text{HI} \wedge t_1 \leq a_i \wedge d_i \leq t_2} c_i^H}{t_2 - t_1}. \tag{2.6}$$

MC Sporadic Task Model. A *task* τ_i characterizes a single piece of code, to be executed *repeatedly* for an indefinite length of time. That is, a *sporadic task* gives rise to a potentially unbounded sequence of jobs — a release is triggered when the corresponding piece of code becomes ready for execution. The period parameter T_i represents the minimum inter-arrival time between any two consecutive job releases (by the same task). Another parameter D_i, denoting the *relative deadline*, is specified for the whole task, that the deadline for each job is its own release plus the D_i value. As a result, the kth job released by task $\tau_i = \{C_i, T_i, D_i, \chi_i\}$ can be represented with the aforementioned 4-tuple model as $J_{i,k} = \{a_{i,k}, C_i, a_{i,k} + D_i, \chi_i\}$, where its release $a_{i,k}$ must be at least T_i time units after the release time of its predecessor $J_{i,k-1}$. We sometimes consider a special set of tasks, with *implicit deadlines* that $D_i = T_i$ for all i.

The utilization of a regular (i.e., non-MC) implicit-deadline sporadic task system denotes the sum of the ratios of the WCETs to periods of all the tasks in the system. We may define analogous concepts for MC sporadic task systems. Let τ denote an MC implicit deadline sporadic task system. Under a dual-criticality system, let $u_i^L = (C_i^L/T_i)$ and $u_i^H = (C_i^H/T_i)$ denote the per-criticality *utilizations* of task τ_i..

For each of x and y in $\{L, H\}$, we define the system-wide *utilization* parameters as follows:

$$U_x^y = \sum_{\tau_i \in \tau \bigwedge \chi_i = x} u_i^y = \sum_{\tau_i \in \tau \bigwedge \chi_i = x} \frac{C_i^y}{T_i}. \tag{2.7}$$

Hence the LO-criticality total system utilization of task-system τ is $\left(U_L^L + U_H^L\right)$, and its HI-criticality total system utilization is U_H^H.

Under Vestal's interpretation, system behavior solely depends on the execution time of each job. As one may tell, the main differences between Vestal Models and traditional real-time workload models (see Sec. 2.1) are: (i) each piece of work has an associated criticality level χ_i, and (ii) a WCET vector is provided instead of a single threshold. During runtime, a level-l task τ_i (or job) may trigger a system-wide mode switch to level $\mathscr{L}$ if its execution exceeds $C_i^{\mathscr{L}}$ and does not signal finishing, where $\mathscr{L} \leq l$.

2.4.3 Related MC Schedulers

The pioneering work on the verification of an MC system was done by Vestal (Vestal, 2007) in 2007, which targets the MC task scheduling problem on a single processor platform with constant speed. It shows that neither rate monotonic (Liu and Layland, 1973) nor deadline monotonic (Leung, 2004) priority assignment is optimal for MC system; however Audsley's algorithm (Audsley, 2001) is found to be applicable.

MC Job Scheduling. Baruah et al. (Baruah et al., 2010a) (Baruah et al., 2012a) show that even scheduling MC job is NP-hard in the strong sense, and an efficient approximation algorithm named *Own-Criticality-Based-Priority (OCBP)* is proposed, with a speedup factor of $(\sqrt{5}+1)/2$ (the golden ratio) and $\Theta(L/\ln L)$, for the two-criticality-level subcase and L-criticality-level cases

($L > 2$), respectively. It is also shown (by an example) that no non-clairvoyant algorithm can achieve a better speedup bound in the dual-criticality case.

MC Task Scheduling on a Uniprocessor. In the sporadic task model, jobs are released recurrently with minimum time gaps (i.e., period). The OCBP algorithm is enhanced to support sporadic systems (Li and Baruah, 2010), with pseudo-polynomial time complexity of offline schedulability test and run-time updating. Guan et al. (Guan et al., 2011) improve this algorithm to a new version named PLRS (Priority-List-Reuse-Scheduling), which updates the *priority list* in $\Theta(n^2)$ time. Baruah et al. (Baruah et al., 2011a) specialize the classic EDF algorithm to support MC systems, by introducing Virtual Deadlines (VD) — the EDF-VD algorithm shrinks deadlines of more important tasks by a common factor, in order for them to preserve enough capacity. The speedup factor of EDF-VD is improved from the original $(\sqrt{5}+1)/2$ to $4/3$ (Baruah et al., 2012b) for the dual-criticality MC task scheduling problem. This speedup result is shown to be *optimal* (Baruah et al., 2012b) in the sense that no non-clairvoyant algorithm can achieve a smaller speedup bound.

MC Task Scheduling on Multiprocessor. The EDF-VD algorithm is extended to cope with multiprocessor platforms in (Li and Baruah, 2012), by applying a previously-proposed multiprocessor *global* scheduling algorithm called fpEDF (Baruah, 2004). For *implicit-deadline* systems, the speedup factor of the global EDF-VD scheduling algorithm is shown to be no larger than $\sqrt{5}+1$, while the *partitioned* version (that assigns each task to a dedicated processor) has a speedup factor of $(8m-4)/(3m)$ for m processors, according to (Li and Baruah, 2012). A *fluid-based* scheduling mechanism named MC-Fluid is recently proposed (Lee et al., 2014) also for scheduling MC implicit-deadline sporadic task systems upon *identical* multiprocessor platforms. It is shown to have a speedup bound no worse than $(\sqrt{5}+1)/2$ for scheduling dual-criticality systems, which is the best known speedup bound result for multiprocessor MC scheduling (prior to our work).

Here we only mentioned work that is closely related to our thesis, while much other prior work on MC scheduling can be found in the review (Burns and Davis, 2016), which has been updated every six months since its first release in 2013. Most existing work only considers uncertainty that

arises from the WCET estimations, while some (e.g., (Baruah, 2012), (Burns and Davis, 2013), (Baruah and Chattopadhyay, 2013)) have considered uncertainty in specifying minimum inter-arrival durations for sporadic tasks. As mentioned at the beginning of this chapter, detailed descriptions of some closely related scheduling strategies will appear in later chapters/sections.

CHAPTER 3: WHEN MC ARISES FROM WCET ESTIMATIONS

Part of the central thesis of this dissertation is to examine the effectivenesses of existing scheduling methods for systems where MC arises from estimations of different aspects. First of all, in this chapter, we will focus on Vestal's interpretation (see Sec2.4.2) of MC schedulability, where the mode switch is solely triggered by the executions of workloads exceeding certain thresholds (i.e., their WCET estimations).

The idea behind Vestal's mixed-criticality model is that the true execution time of each piece of code cannot be known precisely prior to run-time, and must therefore be estimated for system analysis prior to run-time. For MC systems, it may make sense to construct multiple WCET estimations under more or less conservative assumptions (so that we can have greater or lesser levels of assurance that the models do indeed bound the actual run-time behavior of the system). The correctness of the entire system will be validated under the less conservative assumptions; while the correctness of only the more critical parts will be guaranteed under the more conservative estimations. A large body of prior work on MC scheduling focuses on the Vestal model (see, (Burns and Davis, 2016) for an up-to-date review).

In this chapter, we will provide improved results over some state-of-the-art schedulers under various kinds of settings:

- For uniprocessor job scheduling, Sec. 3.1 proposes an algorithm named LE-EDF and shows that it dominates the OCBP algorithm (Baruah et al., 2010b) while out-performs the MC-EDF algorithm (Socci et al., 2013).
- For uniprocessor task scheduling, Sec. 3.2 adds a new parameter to the existing MC task model to better capture the uncertain behaviors. Experimental studies are conducted, showing

that the proposed algorithm under new model results in higher schedulability ratio in most scenarios.

- For multiprocessor task scheduling, the open problem of "What is the best possible speedup?" is answered in Sec. 3.3. We propose an algorithm named MCF, prove its speedup of $4/3$. It has shown that $4/3$ is the best possible speedup for any uniprocessor MC task scheduler. This directly implies our result cannot be further improved for the more general (multiprocessor) case, and thus closes the problem.

We (in this chapter) limit the total number of criticality levels as two, and use HI and LO instead of numbers to denote the criticality levels (with HI being more important).

3.1 MC Job Scheduling on a Uni-Processor

Most of the contributions made in this section can be found at (Guo and Baruah, 2015a).

3.1.1 System Model

We assume a uniprocessor platform with a constant execution speed of 1, upon which a set of Vestal jobs $J = \{J_1, J_2, ..., J_n\}$ is to be scheduled. Each job J_i can be represented by 4-tuple of parameters: $\{a_i, [c_i^L, c_i^H], d_i, \chi_i\}$, where a_i denotes the release time of the job J_i, d_i represents its absolute deadline, $c_i = [c_i^L, c_i^H]$ is the WCET vector ($c_i^L \leq c_i^H$), and $\chi_i \in \{LO, HI\}$ gives its criticality level. According to Sec. 2.4.2, $c_i^L = c_i^H$ if $\chi_i = \text{LO}$.

The interpretation is that the jobs in J are to be executed on a single preemptive processor that has two execution modes: a HI-*criticality* mode and a LO-*criticality* (or *normal*) mode. The system starts out executing at the normal mode, while can switch to the HI-criticality mode any time if a job J_i with $\chi_i = \text{HI}$ has been executed for c_i^L time units, but does not signal its finishing.

A *clairvoyant* scheduling algorithm is one that knows, prior to scheduling an instance, precisely how much execution time each job in the instance will require in order to complete. Here we

assume that a scheduler cannot be clairvoyant; i.e., it is not *a priori* known how much time will each HI-criticality need.

Definition 3.1. *A scheduling strategy for MC job set is* ***correct*** *if it satisfies the following two properties:*

1. *Each job J_i meets its deadline if all jobs complete execution upon having executed for no more than their* LO*-criticality WCETs; and*
2. *Each* HI*-criticality job J_i meets its deadline if all* HI*-criticality jobs complete execution upon having executed for no more than their* HI*-criticality WCETs.*

A scheduling strategy for MC instances is ***partially correct*** *if it satisfies the second item above, but not necessarily the first.*

That is, a correct scheduling strategy ensures the correct execution of HI-criticality jobs provided each HI-criticality job completes upon executing for no more than its HI-criticality WCET. It additionally ensures the correct execution of LO-criticality jobs if each job completes upon executing for no more than its LO-criticality WCET.

Notations. Without loss of generality, we will assume that the HI-criticality jobs in the given MC job set J are indexed $1,2,...,n_h$ and the LO-criticality jobs are indexed $n_{h+1},...,n$, where n_h is the number of HI-criticality jobs. Let $t_1,t_2,...,t_{k+1}$ denote the at most $2n$ distinct values for the release time and deadline parameters of the n jobs, in strictly increasing order (redundancy is eliminated, so $\forall j, t_j < t_{j+1}$). These release time and deadlines partition the whole time duration of interest $[\min_i\{a_i\},\max_i\{d_i\})$ into k *intervals*, which will be denoted as $I_1,I_2,...,I_k$, with I_j denoting the interval $[t_j,t_{j+1})$.

3.1.2 Algorithm LE-EDF

In this subsection, we describe Algorithm LE-EDF[1] for scheduling MC instances that are represented using the model discussed in Sec 3.1.1 above. We will also illustrate, via a running example, the behavior of LE-EDF when scheduling such an MC instance.

The high-level description of our algorithm is as follows. Given an MC job set J, we first construct, prior to run-time, a *scheduling table* that reserves a certain amount of execution time for each HI-criticality job within each time interval $I_j = [t_j, t_{j+1})$, for $1 \leq j \leq k$, in order to ensure that no HI-criticality deadline will be missed even under the most conservative case. To this end, LE-EDF is in some sense similar to the zero-slack technique developed by Niz et al. (Niz et al., 2009), which mainly focused on fixed priority schemes such as rate-monotonic instead of EDF-based ones (which is our focus). To comply with this scheduling table, HI-criticality jobs are divided into *sub-jobs* with different deadlines. Dispatch decisions at run-time are taken in a manner that HI-criticality jobs being executed for at least the amounts mandated in the scheduling table (by having sub-jobs meeting their assigned deadlines), while using the remaining computing capacity to execute LO-criticality jobs. The latest execution (LE) manner in which the sub-job set is constructed is described in Sec 3.1.2.1; run-time dispatching (under EDF) is detailed in Sec 3.1.2.2.

3.1.2.1 Sub-Job Construction (LE)

To construct the scheduling table, we first identify (Step 1 below) the latest time intervals during which the HI-criticality jobs must execute if each were to execute for its HI-criticality WCET; having identified these intervals, we construct (in Step 2) an EDF schedule for the HI-criticality jobs in these intervals.

Step 1. *Considering only the* HI*-criticality jobs in the instance, determine the intervals during which the jobs would execute upon a speed-*1 *processor, if*

1. *each job executes for its* HI*-criticality WCET,*

[1]The two steps, shown in Sec. 3.1.2.1, in the construction of the scheduling table, explain the name given to our algorithm: *Latest Execution times, with EDF* scheduling

2. *each job completes by its deadline, and*

3. *execution occurs as late as possible.*

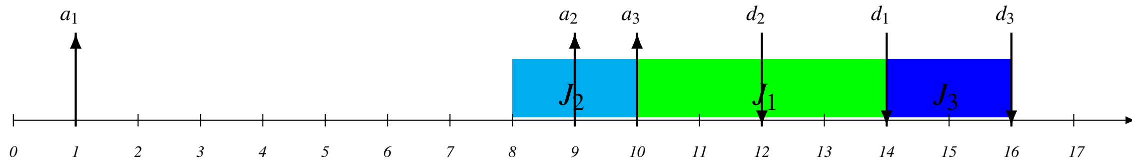
It is evident that these intervals may be determined by filling in the schedule "backward"; i.e., considering the jobs in non-increasing order of their deadlines, and allocating the cumulative execution requirements of these jobs. They can therefore be determined in $\Theta(n_h \log n_h)$ time (which is the complexity of sorting), where n_h denotes the number of HI-criticality jobs. We illustrate this step in Example 3.2 below.

Example 3.2. *Consider the instance consisting of the six jobs J_1—J_6 shown in tabular form in Table 3.1, that is to be implemented upon a preemptive uniprocessor (of speed* 1*).*

J_i	a_i	c_i^L	c_i^H	d_i	χ_i
J_1	1	2	4	14	HI
J_2	9	1	2	12	HI
J_3	10	1	2	16	HI
J_4	0	8	8	10	LO
J_5	1	1	1	12	LO
J_6	12	3	3	16	LO

Table 3.1: An example MC collection of jobs.

Considering only the HI*-criticality jobs J_1–J_3 executing for their* HI*-criticality WCETs on a speed-*1 *processor, the intervals identified in Step 1 are as follows:*

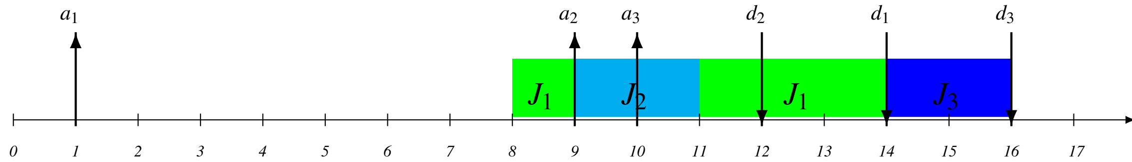


The interval(s) determined in Step 1 are therefore $[8,16)$. *(Observe that in this schedule we are only determining execution intervals, not seeking to determine an actual schedule. Hence the fact that job J_2 seems to be "assigned" execution prior to its release time is irrelevant.)*

Step 2. *Construct an EDF schedule for the* HI*-criticality jobs upon a preemptive processor that has speed* 1 *during the intervals determined in Step 1 above, and speed* zero *elsewhere.*

It follows from the optimality property (Dertouzos, 1974; Liu and Layland, 1973) of EDF that if this step fails to ensure that each HI-criticality job receives an execution amount equal to its HI-criticality WCET prior to its deadline, then no scheduling algorithm can guarantee correctness for this instance. We would therefore ***report failure***: this MC instance is not feasible. The remainder of this section, and Sec 3.1.2.2, assumes that Step 2 above was successful in completing each HI-criticality job prior to its deadline.

Example 3.3. *Consider again the instance of Example 3.2 that is depicted in Table 3.1. In Step 2, the EDF schedule for the* HI*-criticality jobs is constructed only within the intervals identified in Step 1; i.e.,* $[8,16)$[2]*:*



- J_1 *executes during the interval* $[8,9)$ *as the only active job.*
- *Upon release,* J_2 *becomes the earliest-deadline job and is hence allocated execution over the interval* $[9,11)$*, which preempts* J_1 *at* $t = 9$.
- *Upon* J_2*'s completion,* J_1 *executes during the interval* $[11,14)$ *as its deadline is earlier than the only other active job* J_3.
- J_3 *executes in the interval* $[14,16)$ *as the only remaining active job.*

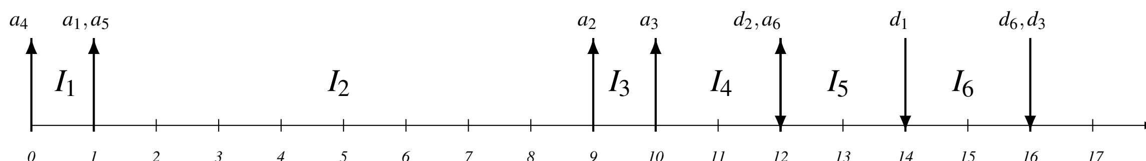
Step 3. *Partition the timeline over* $[\min_i\{a_i\}, \max_i\{d_i\}]$ *(and thus the scheduling table) into the* k *intervals* $I_1, I_2, \ldots, I_k$*. (Recall, from Sec. 3.1.1, that these are the intervals defined by the release*

[2]Note that Step 1 may result in new breakpoints to the timeline and intervals other than release time and deadlines; e.g., $t = 8$.

time and deadlines of all the jobs — LO-criticality and HI-criticality.) For each HI-criticality job J_i and each interval I_ℓ in which it is scheduled in the EDF schedule constructed in Step 2 above, define a **sub-job** *of J_i with the same release time a_i, a WCET equal to the amount of execution that J_i is allocated during Interval I_ℓ, and a deadline equal to $t_{\ell+1}$, the right end-point of Interval I_ℓ.*

By dividing HI-criticality jobs into sub-jobs, and setting proper deadlines for them, they will not be suppressed by LO-criticality jobs in the sense of correctness.

Example 3.4. *For our example instance of Table 3.1, Step 3 partitions the timeline into six intervals* $[0,1)$, $[1,9)$, $[9,10)$, $[10,12)$, $[12,14)$, *and* $[14,16)$.



Each of the HI-criticality jobs is decomposed into the sub-jobs shown in Table 3.2; these are obtained by super-imposing the partitions shown above upon the EDF schedule constructed in Example 3.3.

J_i	a_i	c_i^H	d_i	χ_i
J_{12}	1	1	9	HI
J_{14}	1	1	12	HI
J_{15}	1	2	14	HI
J_{23}	9	1	10	HI
J_{24}	9	1	12	HI
J_{36}	10	2	16	HI

Table 3.2: HI-criticality sub-jobs generated by Step 3 of LE-EDF in Example 3.2.

Counting the number of sub-jobs. Although an individual job in an EDF schedule for an instance of n jobs may be preempted as many as $(n-1)$ times, it is known (see, e.g., (Buttazzo, 2005)) that the *total* number of preemptions in any EDF schedule for an n-job instance cannot exceed $(n-1)$. From this, it follows that the schedule constructed in Step 2 above will contain no more

than $3n_h - 1$ contiguous chunks of execution (here, a $2n_h - 1$ comes from the fact that n_h jobs are being scheduled using EDF, and an additional n_h from the fact that there may be as many as n_h non-contiguous intervals upon which this EDF schedule is executing). Since Step 3 partitions the timeline into no more than $2n - 1$ intervals, it follows that the total number of jobs is bounded from above by $3n_h - 1 + 2n - 1$, which is $\Theta(n)$.

3.1.2.2 Run-Time Scheduling (EDF)

We maintain an EDF (priority) queue during run-time for a combination of the LO-criticality jobs and the HI-criticality sub-jobs (that were constructed during Step 3 above).

We now describe how run-time scheduling decisions are made during the ℓ'th interval I_ℓ, for $\ell = 1, 2, \ldots, k$:

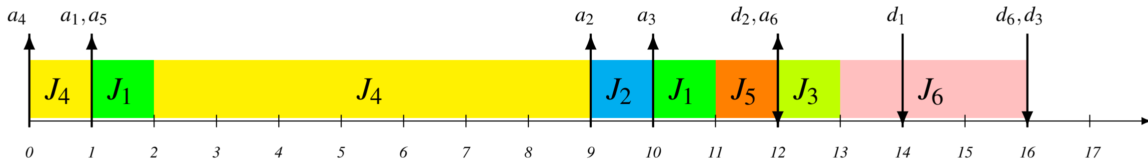
1. We first insert all LO-criticality jobs and HI-criticality sub-jobs that have their release time equal to the start of this interval into the EDF queue.

2. We execute jobs (including sub-jobs) in EDF order, giving HI-criticality sub-jobs higher priority only when tie-breaking jobs with same deadlines. (Note that from the manner in which the sub-jobs are defined, it is guaranteed that all HI-criticality sub-jobs with deadline at the end of this interval complete execution by the end of the interval.)

3. At the end of the interval, all jobs in the LO-criticality EDF queue with deadlines at the end of the interval are *dropped*.

Example 3.5. *We continue scheduling the MC instance considered in Example 3.2 (jobs detailed in Figure 3.1; the* HI*-criticality sub-jobs constructed during Step 3 listed in Figure 3.2). To better illustrate how our algorithm works, we will separately simulate its operation under two different run-time behaviors of the processor.*

§1. *We first consider the case where all* HI*-criticality jobs execute at their* LO*-criticality WCETs. The schedule is depicted in the following figure. (Since sub-job numbers align with interval number, we only label the job numbers.)*



- *For Interval $I_1 = [0,1)$, since no* HI*-criticality sub-job is allocated here, J_4 will be executed as the earliest deadline* LO*-criticality job.*

- *Sub-job J_{12} executes for* 1 *time units at the beginning of Interval $I_2 = [1,9)$. The remaining capacity will be used for jobs with deadline greater than* 9*. As the earliest deadline* LO*-criticality job, J_4 executes first and completes at $t = 9$, after which J_{14} executes over the interval* $[8,9)$ *(and also completes).*

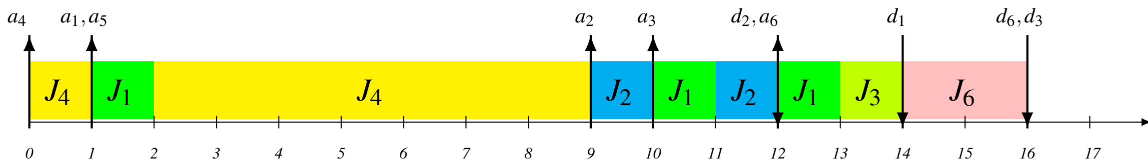
- *Sub-job J_{23} is executed in Interval $I_3 = [9,10)$, and completes at time $t = 10$.*

- *Since all* HI*-criticality jobs execute at their* LO*-criticality WCETs, both J_1 and J_2 are already finished at $t = 10$, and sub-jobs J_{15} and J_{24} require no execution. As a result, the earliest deadline active job (which is J_5) executes over the interval* $[10,11)$*; following by the execution of the only active sub-job J_{36} until $t = 12$.*

- *J_{36} continues its execution in Interval $I_5 = [12,14)$ and finishes by $t = 13$, while the remaining capacity should be used for the only active job J_6.*

- *The only active* LO*-criticality job J_6 executes until it completes at $t = 16$.*

§2. *Now we consider the case where* HI*-criticality jobs J_1 and J_2 execute at their* HI*-criticality WCETs. The schedule is depicted in the following figure.*



- *Execution in Intervals* $I_1 = [0,1)$, $I_2 = [1,9)$, *and* $I_3 = [9,10)$ *remains the same as in the previous case. (Note that although* HI*-criticality job* J_1 *requires more execution now, we do not consider such "upgrade" until all its pre-allocated amounts are finished, which is in Interval* I_4*.)*

- *Both* J_{14} *and* J_{24} *need to complete within interval* $I_4 = [10,12)$*. No capacity remains and the* LO*-criticality job* J_5 *is dropped at its deadline* $t = 12$*.*

- *The interval* $[12,13)$ *is consumed by* J_{15}*. At time* $t = 13$*, there are two active jobs* J_{36} *and* J_6 *with the same deadline, and according to the algorithm, we favor* HI*-criticality jobs in such case, which results in execution of* J_3 *within* $[13,14)$*, and then* J_6 *afterwards.*

Computational complexity. We have seen in Sec. 3.1.2.1 above that Algorithm LE-EDF generates no more than $\Theta(n)$ HI-criticality sub-jobs during the preprocessing phase; during run-time, these sub-jobs are scheduled for execution along with the LO-criticality jobs. We note that standard techniques (see, e.g., (Mok, 1988)) for implementing EDF are known, that allow an EDF schedule for n jobs to be constructed in $\Theta(n \log n)$ time. Consequently, we conclude that the EDF-schedule of Step 2 can be constructed in $\Theta(n_h \log n_h)$ time, and the total scheduler overhead during run-time is also bounded from above by $\Theta(n \log n)$.
Remark. LE-EDF applies for tasks with real number parameters — we restrict the examples with integer time only for easier demonstration and understanding.

3.1.3 Comparison over OCBP

An algorithm named OCBP (for **O**wn **C**riticality **B**ased **P**riorities) was proposed in (Baruah et al., 2010b) for scheduling MC job set, and shown to have a speedup bound of $(\sqrt{5}+1)/2$ (i.e., ≈ 1.618). To date, this is the best speedup bound known for any algorithm for scheduling such MC instances.

We start out briefly describing OCBP. Given an MC instance J, OCBP derives offline a priority ordering for all jobs in the instance, using a variant of the Audsley Optimal Priority Assignment

scheme (Audsley, 2001), in the following manner (here, “scheduling according to priority” means that at each moment in time the highest-priority available job is executed). It determines, as described below, the job that may be assigned lowest priority, and assigns it the lowest priority. This procedure is repeated for the set of jobs excluding the lowest priority one until all jobs are ordered, or at certain iteration a lowest priority job cannot be found.

1. We assign the lowest priority to the LO-criticality job with latest deadline if it would complete by its deadline when every other job were assigned higher priority and execute in their LO-criticality level WCETs.

2. Else, we assign lowest priority to the latest-deadline HI-criticality job if it would complete by its deadline when every other job were assigned higher priority and execute in their HI-criticality level WCETs. Here LO-criticality jobs' HI-criticality level WCETs remains the same as their LO-criticality level WCETs.

3. Else, we declare failure.

The following theorem asserts that any instance that can be scheduled by OCBP is also scheduled by LE-EDF.

Theorem 3.6. *Given any set of MC jobs K, if Algorithm* LE-EDF *fails to complete job(s) at some criticality level on time (either by missing a deadline, or dropping a job), then so will OCBP.*

Proof: There are only two steps during execution at which Algorithm LE-EDF may report a failure to correctly schedule an instance.

If Algorithm LE-EDF fails at Step 1 when constructing schedule table for HI-criticality jobs, it directly follows that there is no *correct* schedule scheme for HI-criticality jobs when they all execute at their HI-criticality WCETs. Thus, OCBP algorithm will also fail to correctly schedule this instance.

Now we consider the case that Algorithm LE-EDF fails during run-time, which indicates that some LO-criticality job J_i missed its deadline and will be dropped at time $t = d_i$. We will show that

OCBP algorithm must also drop a LO-criticality job at or before this time in order to guarantee the correctness of HI-criticality job execution.

One sub-case is that a HI-criticality job has executed longer than its LO-criticality WCET before time $t = d_i$. OCBP algorithm will immediately drop remaining LO-criticality jobs when it occurs, which is at or before time $t = d_i$.

The other sub-case is that all HI-criticality jobs have executed no longer than their LO-criticality WCETs (so far). We will prove by contradiction that OCBP algorithm will also drop some job at or before time $t = d_i$.

Assume OCBP algorithm has not dropped any job at or before time $t = d_i$, which means that it makes so far all jobs meet their deadlines at time $t = d_i$. Consider only the LO-criticality jobs, from the description we can easily tell that both algorithms execute them at an EDF order: OCBP generates the priority list by considering jobs in each criticality level in non-increasing order of deadlines, while LE-EDF uses the remaining capacity for all LO-criticality jobs in the EDF order (after pre-allocating and slicing HI-criticality jobs). Since LE-EDF fails to meet some LO-criticality job's deadline at d_i while OCBP does not, it must be the case that OCBP executes LO-criticality jobs (totally) between time $t = 0$ and $t = d_i$ for a longer time than LE-EDF. Thus OCBP has executed less amounts of HI-criticality jobs until time $t = d_i$ than LE-EDF [3]. However, LE-EDF guarantees that at this deadline d_i, all HI-criticality sub-jobs with deadlines on or before d_i are "must to be finished", which means that a shorter accumulated execution time to HI-criticality jobs will cause a deadline miss in the future. This contradicts the correctness guarantee to HI-criticality jobs of OCBP, and indicates that our assumption that OCBP algorithm has not dropped any job at or before time $t = d_i$ is incorrect. The theorem results from this contradiction. □

Lemma 3.7. *There exists a job set that* LE-EDF *can provide a correct schedule, while OCBP fails to do so.*

[3] Both algorithms have exactly the same idleness periods since (by definition) both will idle the processor only when there is no *active* job.

Proof: Consider the job set in Table 3.1. It has been shown in previous subsections that LE-EDF can correctly schedule this set. However, this instance is not OCBP-schedulable: after assigning J_6 the lowest priority, no job can be further assigned the second lowest priority. □

Lemma 3.7 in conjunction with Theorem 3.6, allows us to conclude that LE-EDF *dominates* OCBP.

3.1.4 Comparison over MCEDF

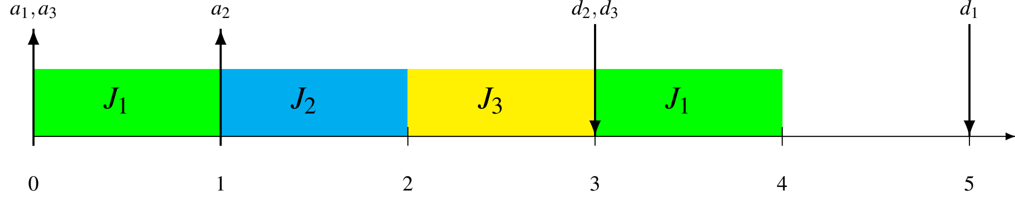
An algorithm named MCEDF was recently (Socci et al., 2013) presented for scheduling MC instances upon processors that are speed bounded by a constant during run-time – i.e., the same kind of workload scheduled by OCBP – and shown to strictly dominate OCBP (to the best of our knowledge, MCEDF is the only algorithm proven to dominate OCBP). We do not yet know whether our algorithm LE-EDF dominates MCEDF or not; we do, however, show below that the converse cannot be the case.

Theorem 3.8. *There are MC instances correctly scheduled by Algorithm* LE-EDF *that MCEDF does not schedule in a correct manner.*

Proof: We present one such instance:

J_i	a_i	c_i^L	c_i^H	d_i	χ_i
J_1	0	2	3	5	HI
J_2	1	1	2	3	HI
J_3	0	1	1	3	LO

It was shown in (Socci et al., 2013) that this instance is not MCEDF schedulable. The following schedule shows how LE-EDF schedules this instance, and correctness is thus verified from this schedule.



□

3.1.5 Experimental Comparisons

We also performed some *simulation experiments* to complement the theoretical conclusions of the theory results. The experimental setup is as described in (Socci et al., 2013, Sec. IV). We generate a large number of MC instances of 20 jobs each according to the following steps.

3.1.5.1 MC Job Generator

Each randomly-generated MC instance is characterized by four parameters:

1. n, the total number of jobs in the instance.

2. u_{all}, a measure of the computational load of the instance. This is equal to the sum of the WCETs of all the jobs in the instance, normalized by the duration of time spanned by their scheduling windows[4].

3. γ, the expected fraction of jobs that are of HI criticality.

4. ζ, the expected number of jobs with scheduling windows that overlap (cover) each time instant. A value $\zeta = 1$ suggests that there are no overlaps between the scheduling windows of any pair of jobs, while $\zeta = n$ means that all jobs have the same release date and deadline).

With values specified for these four parameters, the individual jobs composing the instance are generated randomly according to the following steps.

§1: Release dates. We model job arrivals by a (memoryless) Poisson process. I.e., we generate $(n-1)$ independent and identically distributed random variables x_i according to the exponential

[4]The *scheduling window* of a job is the duration between the job's release time and its deadline.

distribution with $\lambda = 1$. The first job is assigned release date zero ($a_1 := 0$); subsequent release dates are assigned values as $a_{i+1} := a_i + x_i$.

§2: Deadlines. We follow the procedure suggested in (Baruah et al., 2011b) and model relative deadlines (the duration between release date and deadline) as independent and identically distributed random variables drawn from the log-uniform distribution (exponential of uniform distribution $U[b_l, b_u]$).

To obtain the desired values we chose $b_l \leftarrow 0$ and b_u to be the solution to the equation $e_u^b - \zeta b_u - 1 = 0$ (the equation is solved numerically using the Newton-Raphson method), so that expectation for the log-uniform distribution is $E(c) = (e^{b_u} - e^{b_l})/(b_u - b_l) = \zeta$. Since in expectation, a job is released every $(\lambda = 1)$ time unit[s], and will have a scheduling window of duration ζ time units, the expected number of jobs with scheduling windows covering each time instant approaches ζ with increasing n.

§3: Criticality Level. Each job is assigned criticality HI with probability γ (and hence, criticality LO with probability $(1-\gamma)$).

§4: Worst Case Execution Time (WCET). Once all the release dates and deadlines have been assigned, we can determine the total duration of time covered by all the jobs' scheduling windows — this is equal to the latest deadline *minus* the duration of those intervals that do not lie within any scheduling window. Let L_{act} denote this duration. The parameter u_{all} characterizing this workload now determines the cumulative WCETs of all the jobs: $\sum_i c_i = \sigma := u_{\text{all}} L_{\text{act}}$.

An additional straightforward restriction on the WCET of each job is that it cannot exceed the relative deadline of the job. Let d_i' denote the relative deadline of the i'th job. Our method generates WCET one by one in increasing order of relative deadline: In the generation of the i'th WCET c_i, given $c_1, \ldots, c_{i-1}$, the following two inequalities may provide a tighter bound:

$$
\begin{aligned}
c_i &\geq \sigma - \sum_{j=1}^{i-1} c_j - \sum_{j=i+1}^{n} d_j' \\
c_i &\leq \sigma - \sum_{j=1}^{i-1} c_j.
\end{aligned}
$$

It is evident that if either of these equations is violated, the sum of all the WCETs will not equal σ no matter what values the remaining c_j take in their respective ranges $[0, d'_j], j = i+1, ..., n$.

Thus for each of $i = 2, ..., n-1$, the bound for generating the WCET should be

$$\begin{aligned} c_i \geq lb(c_i) &:= \max\{0, \sigma - \sum_{j=1}^{i-1} c_j - \sum_{j=i+1}^{n} d'_j\} \\ c_i \leq ub(c_i) &:= \min\{d'_i, \sigma - \sum_{j=1}^{i-1} c_j\} \end{aligned}$$

The bound of c_1 is simpler, with $lb(c_1) = \max\{0, \sigma - \sum_{j=2}^{n} d'_j\}$ and $ub(c_1) = d'_1$; and c_n is set equal to $\sigma - \sum_{j=1}^{n-1} c_j$. Note that we will only discuss how to randomly generate $c_1, ..., c_{n-1}$ properly in the following, thus i will only take values from 1 to $n-1$.

Although we have determined upper and lower bounds on each c_i value, we cannot simply choose the c_i's uniformly in the calculated range $[lb(c_i), ub(c_i)]$. In order to ensure an unbiased random generation, the expectation (i.e., the mean value) of each WCET needs to be fixed, and may not be $(lb(c_i) + ub(c_i))/2$. Here we assume the sum of the WCETs, which equals σ, is to be shared "fairly" according to relative deadlines. In this context, fairness would dictate that the jobs with longer relative deadline d'_i gets a relatively larger expectation of WECT c_i. More precisely, we desire that the expected values $E(C_i)$ of the WCET's – the c_i values – satisfy

$$E(c_i) = \sigma \times \left(d'_i / (\sum_{i=1}^{n} d'_i) \right)$$

We have chosen the beta distribution to generate these random values c_i within the computed ranges $[lb(c_i), ub(c_i)]$ and the desired expected value $E(c_i)$. One the parameters of beta distribution is fixed to be $\alpha(c_i) = 2$, and the other is given by

$$\beta(c_i) = 2 \times (\frac{ub(c_i) - E(c_i)}{E(c_i) - lb(c_i)})$$

Since the beta distribution generates random values over $[0, 1]$ with expectation value of $\alpha/(\alpha + \beta) = (E(c_i) - lb(c_i))/(ub(c_i) - lb(c_i))$, we need to scale the values into the ranges $[lb(c_i), ub(c_i)]$

by multiplying by $(ub(c_i) - lb(c_i))$ and adding $lb(c_i)$. This ensures that the expectation of c_i is $(E(c_i) - lb(c_i))/(ub(c_i) - lb(c_i)) \times (ub(c_i) - lb(c_i)) + lb(c_i)$ which is equal to $E(c_i)$ as desired.

The Matlab code of the generator is available at:

`http://www.cs.unc.edu/~zsguo/files/MC_job_creator_R.zip`.

3.1.5.2 Schedulability Comparison.

In the experiments of this section, the parameters $\ell_{\text{LO}}(\cdot)$ and $\ell_{\text{HI}}(\cdot)$ (see Def 2.6) of the generated instances range from 0 to 1, with step 0.01. Only "overloaded" instances – those satisfying $\ell_{\text{LO}}^2(J) + \ell_{\text{HI}}(J) > 1$ – are considered since all three algorithms are successful in scheduling the non-overloaded ones.

Among the 33511 successfully generated instances, OCBP fails to schedule 5076 ($\approx 15.1\%$). From amongst these[5], MCEDF reports failure as well for 1986 ($\approx 5.9\%$), only 109 ($\approx 0.3\%$) of which are also unschedulable by LE-EDF. Further, all the instances scheduled by MCEDF (and OCBP) were also scheduled by LE-EDF.

Figure 3.1 depicts the schedulability results for the three algorithms. Instances with similar ℓ_{LO} and ℓ_{HI} values are put into a same small block (with a typical size of 10 to 15 instances). The color of each small block represents the percentage of schedulable sets.

In all these and several other experiments not discussed here, we have not been able to identify any instance that can be scheduled by MCEDF but not by LE-EDF. Although this certainly does not constitute formal proof that LE-EDF dominates MCEDF, it seems clear that generally speaking, LE-EDF is superior to the other two existing algorithms, both in terms of schedulability (as shown in the experiments), and run-time complexity (theoretically shown to be $\Theta(n \log n)$, where $n = |J|$, which is asymptotically much better than OCBP and MCEDF's $\Theta(n^2 \log n)$).

[5]There are no instances scheduled by OCBP but not MCEDF — this is as expected since MCEDF was shown (Socci et al., 2013) to dominate OCBP.

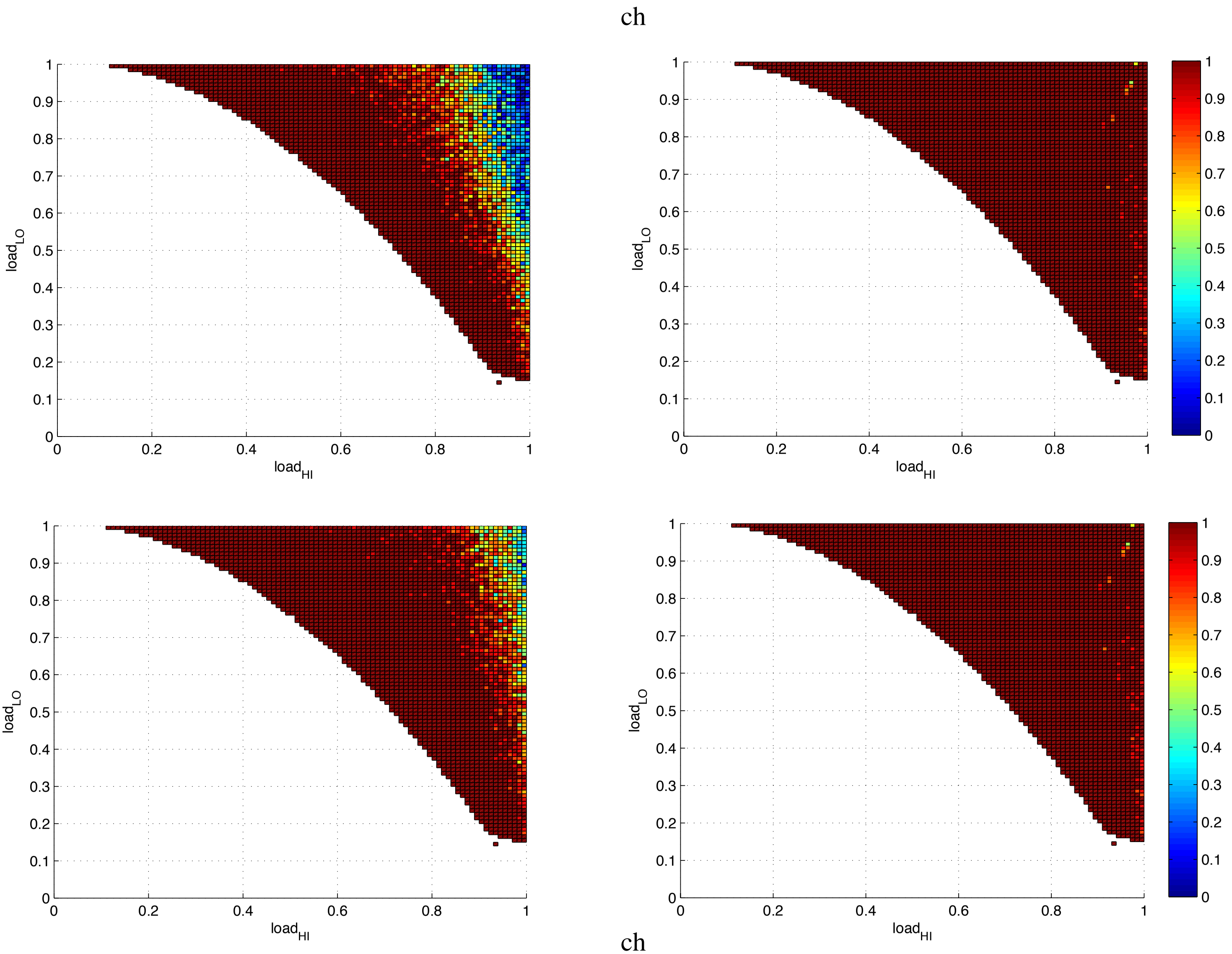


Figure 3.1: Schedulability comparison of OCBP (upper-left), MCEDF (lower-left), and LE-EDF (right, duplicated for easy comparison), where the color of each block represents the fraction of schedulable instances with ℓ_{LO} and ℓ_{HI} parameters falling within certain small ranges. (Informally, red is better – observe that there is almost no blue segment for LE-EDF – please view upon a color monitor/printout)

3.2 MC Task Scheduling on Uni-Processor

Much work has been done on scheduling MC sporadic task set (with implicit deadlines) upon a uniprocessor platform, and algorithms like EDF-VD (Baruah et al., 2012b) has shown to have an *optimal* speedup of $4/3$ (i.e., the speedup cannot be further improved for any non-clairvoyant[6] algorithm). Nevertheless, in this section, we show that by extending the current Vestal model, a new algorithm can be derived with better experimental performance comparing to EDF-VD.

More precisely, this section introduces a new parameter to each dual-criticality MC task that represents the distribution information about its WCET and provides schedulability analysis with respect to the given safety certification requirement of the whole system, which is the *permitted system failure probability per hour (FPH)*. As stated above, dual-criticality tasks are traditionally characterized with two WCET estimations — a LO-WCET and a larger HI-WCET. Our contributions are as follows:

- We propose a supplement to current MC task models: an additional parameter for each HI-criticality task, denoting the *probability* of no job of this task exceeding its LO-WCET *within an hour* of execution.
- We further generalize our notion of system behavior by allowing for the specification of FPH, denoting an upper bound on the probability that the system may fail to meet its timing constraints during any hour of running.
- We derive a novel scheduling algorithm (and an associated sufficient schedulability test) for a given MC task set and an allowed system FPH. We seek to schedule the system such that the probability of failing to meet timing constraints during run-time is guaranteed to be no larger than the specified allowed system FPH.

We emphasize that our algorithm, in the two criticality level case, requires just one probabilistic parameter per task — the probability that the actual execution requirement will exceed the specified

[6] A *clairvoyant* scheduler has the privilege of knowing the system execution behavior, e.g., whether a task τ_i would signal its finishing when being executed for c_i^l time units, for any l, prior to run-time.

LO-WCET in an hour. We believe our scheduling algorithm is novel in that it is, to our knowledge, the first MC scheduling algorithm that makes scheduling decisions (e.g., when to trigger a mode switch) based not only on release dates, deadlines, and WCETs, but also on the probabilities drawn from probabilistic timing analysis tools (see (Cazorla et al., 2013) (Hansen et al., 2009) (Cucu-Grosjean et al., 2012) for examples of such tools). Most of the contributions made in this section can be found at (Guo et al., 2015).

3.2.1 Motivation and Prior Work

Safety-critical systems are failure prone as any other system, and today's system certification approaches recognize this and specify *permitted system FPHs*. The underlying idea is to certify considering more realistic system models which account for any possible behavior, included faulty conditions, and the probability of these behaviors occurring. The gap that still exists is between such enhanced models and the current conservative deterministic analyses which tend to be pessimistic.

The *worst-case execution time* (WCET) abstraction plays a central role in the analysis of real-time systems. The WCET of a given piece of code upon a specified platform represents an upper bound to the duration of time needed to finish execution. Unfortunately, even when severe restrictions are placed upon the structure of the code (e.g., known loop bounds), it is still extremely difficult to determine the absolute WCET. An illustrative example is provided in (Souyris et al., 2005), which demonstrates how the simple operation "`a = b + c`" on integer variables could take anywhere between 3 and 321 cycles upon a widely-used modern CPU. The number of execution cycles highly depends on factors such as the state of the cache when the operation occurs. WCET analysis has always been a very active and thriving area of research, and sophisticated timing analysis tools have been developed (Wilhelm et al., 2008).

Traditional rigorous WCET analysis may lead to a result of much pessimism, and the occurrence of such WCET is extremely unlikely, unless under highly pathological circumstances. For instance, although a conservative tool would assign the "`a = b + c`" operation a WCET bound of 321 cycles, a less conservative tool may assign it a much smaller WCET (e.g., 30) with the understanding that

the bound may be violated on rare occasions under certain (presumably highly unlikely to occur) pathological conditions.

Under the current mixed-criticality model, it is assumed that *all* HI-criticality jobs may require executions up to their HI-WCETs in HI mode simultaneously. EDF-VD is a well-known MC scheduler with optimal speedup bound under such an assumption.

Overview of EDF-VD (Baruah et al., 2011a)(Baruah et al., 2012b). Given a set of dual-criticality tasks $\tau = \{\tau_1, ..., \tau_n\}$ to be scheduled on a unit-speed processor, EDF-VD computes the shrinking factor x as $x \leftarrow U_H^L/(1-U_L^L)$, and checks whether $xU_L^L + U_H^H \leq 1$. If so, it sets virtual deadlines $\hat{T}_i$ for each HI-criticality task τ_i as $\hat{T}_i \leftarrow xT_i$; and if not, it declares failure. Run-time scheduling is done according to EDF order with virtual deadlines under normal mode. If some job does execute beyond its LO-criticality WCET without signaling that it has completed execution, then all LO-criticality jobs are immediately discarded, and HI-criticality tasks continue execution according to EDF with their actual job deadlines (instead of virtual ones).

However, since WCET tools are normally quite pessimistic, LO-WCET are not very likely to be exceeded during runtime.

Example 3.9. *Consider a system composed of two* independent[7] HI-*criticality tasks* τ_1 *and* τ_2*, where each task is denoted by two utilization estimations* $u^L \leq u^H$*. The two tasks* $\tau_1 = \{0.4, 0.6\}$*,* $\tau_2 = \{0.3, 0.5\}$*, represented by utilizations in different modes, are to be scheduled on a preemptive unit-speed uniprocessor. It is evident that this system cannot be scheduled correctly under the traditional model, since the* HI-*criticality utilization, at* $(0.6+0.5)$*, is greater than the processor capacity which is* 1.

However, suppose that: (i) absolute certainty of correctness is not required; instead it is specified that the system FPH should not exceed 10^{-6}*; and (ii) it is known that the timing analysis tools used to determine* LO-*criticality WCETs ensure that the likelihood of any job of a task exceeding its* LO-*WCET is no larger than* 10^{-4} *per hour. Based on the task independence assumption, the probability of jobs from both tasks exceeding their* LO-*WCETs is* $10^{-4} \times 10^{-4} = 10^{-8}$ *per hour.*

[7] Two events are independent if the occurrence of one event does not have any impact on the other.

Thus, we know that it is safe *to ignore the case that both tasks simultaneously exceed their* LO-*WCETs. Hence, the system is* probabilistically feasible*, since the total remaining utilization will not exceed:* $\max\{0.4+0.3, 0.4+0.5, 0.6+0.3\} = 0.9 \leq 1$.

Example 3.9 gives us an intuition that with the help of probabilistic analysis, we may be able to ignore some extremely unlikely cases, and come up with some *less pessimistic* schedulability analysis — if we have the prior knowledge that there will be at most a fixed number of HI-criticality tasks with execution exceptions per hour, dropping of less important jobs may not be necessary at all.

Schedulability with Probabilities. In order to formally describe the uncertainty of the WCET estimations and overcome the over-pessimism, many attempts in introducing probability to real-time system model and analysis have been made.

Edgar and Burns (Edgar and Burns, 2001) made a major step forward in introducing the concept of *probabilistic confidence* to the task and the system model. Their work targets the estimation of probabilistic WCETs (pWCETs) from test data for individual tasks, while providing a suitable lower bound for the overall confidence level of a system. Since then, on one hand much work has been done to provide better WCET estimations and a predicted probability of any execution exceeding such estimation alongside the usage of extreme value theory, e.g., (Hansen et al., 2009) (Griffin and Burns, 2010) (Cucu-Grosjean et al., 2012). In static probabilistic timing analysis, random replacement caches are applied to compute exact probabilistic WCETs, and probabilistic WCET estimations with preemptions, (Davis et al., 2013). More recently, researchers have initiated some pWCET estimation studies (Slijepcevic et al., 2013) (Hardy and Puaut, 2013) in the presence of permanent faults and disabling of hardware elements. On the other hand, there is only one piece of work which proposes probabilistic Execution Time (pET) estimation (David and Puaut, 2004) based upon a tree-based technique. The pET of a task describes the probability that the execution time of the job is equal to a given value, while the pWCET of a task describes the probability that the worst-case execution time of that task does not exceed a given value.

Based upon the estimated pWCET and pET parameters (often as distributions with multiple values and associated probabilities), studies aim to provide estimations that the probability of missing a deadline of the given system is small enough for safety requirements; e.g, of the same order of magnitude as other dependability estimations. Tia et al. (Tia et al., 1995) focus on an unbalanced heavy loaded system (with maximum utilization larger than 1 and much smaller average utilization) and provide two methods for probabilistic schedulability guarantees. Lehoczky (Lehoczky, 1996) proposes the first schedulability analysis of task systems with probabilistic execution times. This work is further extended to specific schedulers, such as earliest deadline first (EDF, (Liu and Layland, 1973)) in (Zhu et al., 2002) and under fixed priority policy in (Gardner and Liu, 1999). (Díaz et al., 2002) provides a very general analysis for probabilistic systems with pWCET estimations for tasks. In addition to WCET estimations, statistical guarantees are performed upon the minimum inter-arrival time (MIT) estimation as well (Abeni and Buttazzo, 1999) (Maxim and Cucu-Grosjean, 2013). Schedulability analysis based on pETs (instead of pWCETs) is also done in (Hansen et al., 2002) for limited priority level case (quantized EDF), and in (Manolache et al., 2004) where an associated schedulability analysis on multiprocessors is presented. Statistical response-time analysis, e.g., (Lu et al., 2012), can be further done to real-time embedded systems based upon the probabilistic schedulability analysis.

Unfortunately, most existing studies have only shown probabilistic schedulability analysis (e.g., estimating the likelihood for a system to miss any deadline) or probabilistic response time analysis to existing algorithms such as EDF and fixed priority scheduling, instead of incorporating probabilistic information into the scheduling strategy. In other words, current research has not addressed the possibility of making *smarter scheduling decisions* with probabilistic models from existing powerful probabilistic timing analysis tools (e.g., (Bernat et al., 2003)) that provide WCET bounds and specified confidences. To our best knowledge, there is only one piece of work presenting scheduling algorithms for probabilistic WCETs of tasks described by random variables (Maxim et al., 2011), which extends the optimality of Audsley's approach (Audsley, 2001) in fixed-priority scheduling to the case WCETs are described by distribution functions.

Finally, none of the existing schedulability analysis work regarding mixed-criticality considered pWCET. Since the major goal of both mixed-criticality and introducing probability are the same, which is to better deal with the over-pessimism of running time estimations, we believe a model that considers both aspects would lead us to much more promising results in real-time system design and verification.

3.2.2 Model

We start out considering a workload model consisting of *independent implicit-deadline sporadic tasks*, where the deadline and the period of a task share the same value (in contrast to constrained-deadline ones). Throughout this section, an integer model of time is assumed — all task periods are assumed to be non-negative integers, and all job arrivals are assumed to occur at integer instants in time.

In traditional MC models, each HI-criticality task is characterized by two WCETs, C^L and C^H, which could be derived with different timing analysis tools. By the level of pessimism and/or other properties in the timing analysis, such a tool usually provides a confidence for its resulting WCET estimates. However, no prior work on MC analysis has leveraged any information from the confidence of the provisioned WCET.

Existing MC analysis usually makes the most pessimistic assumption that *every* HI-criticality task may execute beyond its LO-WCET and reach its HI-WCET *simultaneously*.

In real applications, the industry standards usually only require the expected probability of missing deadlines within a specified duration to be below some specified small value, as the deadline miss can be seen as a faulty condition. Instead, our work aims at leveraging probabilistic information from the timing analysis tools (i.e. confidence) to rule out the too pessimistic scenarios and to improve schedulability of the whole system under a probabilistic standard.

Our work also differs from most prior work on WCET analysis as follows. Existing timing analysis work usually analyzes the WCET for a task on a per-job basis; i.e., by focusing on the distribution of WCETs of jobs of a certain task. When it comes to analyzing a series of consecutive

jobs generated from the same task, the distribution is directly applied. It is usually assumed that i) all jobs WCET of a certain task obey the same distribution (identically distributed), and ii) the WCET of a job is probabilistically drawn from the distribution with no dependence on other jobs of the same task (independence).

While the independence assumption holds for the worst-case execution time, as we will see in Sec. 3.2.3, it may not hold for the task execution time. For example, in many applications such as video frames processing, the execution times of processing consecutive frames of a certain video are usually dependent. However, the event that a certain task has ever overrun its provisioned execution time in time intervals of a certain adequate large length (e.g., an hour) is independent from the scenario in other such intervals, and the probability of such event should be derived from the confidence of corresponding timing analysis tools only.

Before detailing our task model, few statistical notions need to be introduced in order to clarify previous and next observations. Given a task τ_i, its pWCET estimate comes from a random variable (the worst-case execution time distribution), notably continuous distributions[8] denoted by $\mathscr{C}_i$. Equivalent representations for distributions are the probabilistic density functions (pdfs), $f_{\mathscr{C}_i}$, the Cumulative Distribution Functions (CDFs) $F_{\mathscr{C}_i}$, and the Complementary Cumulative Distribution Functions (CCDFs), $F'_{\mathscr{C}_i}$. In the following, calligraphic uppercase letters are used to refer to probabilistic distributions, while non-calligraphic letters are used for single value parameters.

The CCDF representation relates *confidence* to *probabilities*; indeed, from $F'_{\mathscr{C}_i}(C^L)$ we have the probability of exceeding C^L. *The confidence is then for C^L being an upper-bound to task execution time*. The WCET threshold, simply named pWCET or WCET in the rest of the section, is a tuple $\langle C^L, p(\textsc{lo})\rangle$, where the probability $p(\textsc{lo})$ sets the confidence (at the job level) of exceeding C^L, $p(\textsc{lo}) = F'_{\mathscr{C}}(C^L) = P(\mathscr{C} > C^L)$. By decreasing the probability threshold $p(\textsc{lo})$, thus, the confidence on the upper-bounding worst-case, C^L increases.

[8]The timing analysis that makes use of the extreme value theory, by definition provides continuous distributions as pWCET estimates,(Cazorla et al., 2013); they are then discretized, to ease their representation, by assigning them a discrete support.

Given A the event that a job exceeds its threshold and $p^A = P(\mathscr{C}_i > C^L)$ its probability of happening; given B the event that another job exceeds its threshold (in a different execution interval) with $p^B = P(\mathscr{C}_i > C^L)$ its probability of happening. With separate jobs as well as separate execution intervals, and considering WCETs, the conditional probability $P(A|B)$ is equal to $P(A)$, thus, the joint probability is

$$P(A,B) = P(A|B) \times P(B) = P(A) \times P(B), \tag{3.1}$$

due to the independence between WCETs. Projecting the per job probability threshold $p(\textsc{lo}) = F'_{\mathscr{C}_i}(C^L)$ to one-hour task execution interval, we make use of the joint probability of all the exceeding threshold events within the one-hour interval. The joint probability is

$$P(\mathscr{C}_i > C^L, \mathscr{C}_i \leq C^L, \mathscr{C}_i \leq C^L, \dots, \mathscr{C}_i \leq C^L), \tag{3.2}$$

as the probability of just a task job exceeding its thresholds C^L, and all the others not exceeding C^L. With full independence, the probability of exceeding the threshold in one hour would be at most $1 - F_{\mathscr{C}_i}(C^L) \times \lfloor T_i/3,600,000 \rfloor$, with the task τ_i period T_i expressed in *msec*.

3.2.3 Probabilistic Schedulability

In our model, an *allowed system FPH* F_S is specified. It describes the permitted probability of the system failing to meet timing constraints during one hour of execution F_S may be very close to zero (e.g., 10^{-12} for some safety critical avionics functionalities).

A *failure probability* parameter f_i can be added to the HI-criticality tasks. f_i denotes the probability that the actual execution requirement of *any* job of a HI-criticality task τ_i exceeds C_i^L (but still below C_i^H) in one hour (i.e., the adequate long time interval we assumed in this section). f_i depends on a failure distribution $F_i(t)$ that describes the task τ_i probability of failure (at least) up to and including time t. Since $F_i(t)$ would refer to time (interval) and to task execution, it is going to

be the one we computed for one-hour interval or any another interval, Equation ((3.2)). Thus, f_i is directly derived from $F_{\mathscr{C}_i}$.

Thus, a HI-criticality task is represented in our model by four parameters: $\tau_i = ([C_i^L, C_i^H], f_i, T_i, \chi_i)$; LO-criticality tasks continue to be represented by three parameters as before. This enhanced model is essentially asserting, for each HI-criticality task τ_i, within a time interval of one hour, no job of τ_i has an execution greater than C_i^H and the probability of *any* job of τ_i has an execution greater than C_i^L is $f_i > 0$ — we would expect f_i to be a very small value. In our work we assume C_i^H the deterministic WCET, $\langle C_i^H, 0\rangle$, while $\langle C_i^L, f_i > 0\rangle$ the probabilistic WCET with $C_i^L \leq C_i^H$. Normally we do not guarantee higher assurance for LO-criticality tasks (than HI-criticality ones), and thus only C_i^L are adopted for them.

Definition 3.10 (MC Task Instance)**.** *An MC task instance I is composed of an MC task set* $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ *and a system failure requirement* $F_S \in (0,1)$. *(Although* F_S *may be arbitrarily close to* 0, $F_S = 0$ *is not an acceptable value — "nothing is impossible.")*

Let $n_{\text{HI}} \leq n$ denote the number of HI-criticality tasks in τ. We assume that the tasks are indexed such that the HI-criticality ones have lower indices; i.e., the HI-criticality tasks are indexed $1, 2, \ldots, n_{\text{HI}}$.

We seek to determine the ***probabilistic schedulability*** of any given MC task instance:

Definition 3.11 (probabilistic schedulability)**.** *An MC task set is* strongly probabilistic schedulable *by a scheduling strategy if it possesses the property that upon execution, the probability of missing any deadline is less than* F_S*. It is* weakly probabilistic schedulable *if the probability of missing any* HI*-criticality deadline is less than* F_S*. (In either case, all deadlines are met during system runs where no job exceeds its* LO*-WCET.)*

That is, if a schedulability test returns strongly schedulable, then all jobs meet their deadlines with a probability of no less than $1 - F_S$, while weakly schedulable only guarantees (with probability no less than $1 - F_S$ that) HI-criticality jobs meet their deadlines. Moreover, similar to all MC works, for either strongly or weakly probabilistic schedulable, all deadlines are met when all jobs

finish upon executing their LO-WCETs. Again, F_S comes from the natural need of some system certifications, while f_i is the additional information for each task that we need to derive from WCET estimations to achieve such probabilistic certification levels.

3.2.3.1 On the WCET Dependencies

In our model, the FPH of each task f_i represents the probability of *any* job of the task τ_i exceeding its LO-WCET. Thus, dependences between tasks and task executions could have a strong impact on f_i. We hereby detail how we intend to cope with statistical dependence.

In (Cucu-Grosjean, 2013) it has been shown that *neither* probabilistic dependence among random variables *nor* statistical dependence of data implies the loss of independence between tasks' pWCETs or WCET estimates. The WCET is an upper-bound to any execution time, and it embeds all the dependence effects. This makes the important consequence of the independence between WCETs: jobs and tasks modeled with WCETs are independent because WCETs already embed dependence effects.

In our MC study, the LO-WCET may come from consideration of the execution time rather than of the WCET. Although both execution bounds (LO-WCET and HI-WCET) are so far called worst-case execution time estimations, the LO-WCET may also serve as an execution time upper-bound, where dependence between tasks and within tasks needs to be more carefully accounted for.

Each MC task may generate an unbounded number of jobs. Since jobs generated from the same task set typically represent the execution of the same piece of code, the failure probability f_i of a task τ_i represents the likelihood that the required execution time of *any* job generated within an hour by τ_i will exceed C_i^L. In (Santinelli et al., 2014; Melani et al., 2013) it has been showed that real safety-critical embedded systems have natural variability on the task execution time, thus it is reasonable to assume independence or extremal independence between jobs.

Concerning task dependencies, we can cope with the dependence by specifying the task pairwise dependence model. Assuming we are given a list of pairs (τ_i, τ_j) indicating that (WC)ETs of these two tasks may be dependent on each other. It means that the probability of them both

exceeding their LO-WCET is no longer the product of their individual probabilities. By knowing $P(\mathscr{C}_i > C_i^L, \mathscr{C}_j > C_i^L)$ we are able to model (τ_i, τ_j) dependence including execution time task dependencies in our framework, Sec. 3.2.4. However, it is reasonable to assume that many (or most) task pairs do not have such dependencies to each other (although at the execution time level), since the limited impact of one task to another in a mixed-critical partitioned system. Furthermore, it is worthy to note that execution times are observed with other tasks executing in parallel, thus, the execution time measuring embeds already dependence effects from other tasks. In future work, we will explain better task dependence modeling at run-time.

To resume, the dependence between jobs of the same task and between tasks are covered by our model.

3.2.3.2 Utilization Costs

The notion of *additional utilization cost*, defined below, helps quantify the capacity that must be provisioned under HI-criticality mode.

Definition 3.12 (additional utilization cost)**.** *The* additional utilization cost *of* HI*-criticality task* τ_i *is given by*

$$\delta_i = (C_i^H - C_i^L)/T_i. \tag{3.3}$$

Since we consider EDF schedulability instead of fixed priority, we would like to know whether, and *how likely* system utilization may exceed 1: (i) if it is extremely unlikely that the total HI-criticality utilization exceeds 1 (weakly probabilistic schedulable), we could assert a system that is infeasible in traditional MC model to be probabilistic feasible; (ii) if it is extremely unlikely that total system utilization exceeds 1 (strongly probabilistic schedulable), we could decide not to drop any LO-criticality task even if some HI-criticality tasks accidentally suffer from failures (that they require more execution time than expected).

Example 3.9 has shown an infeasible task set (under traditional MC scheduling) being weakly probabilistic schedulable under our model. As seen from the definitions, existing mixed-criticality systems are often analyzed under two modes — the HI mode and the LO mode, and mode switch is

triggered when any HI-criticality job exceeds its LO-WCET without signaling finishing. Upon such a mode switch, deadlines of all LO-criticality jobs will no longer be guaranteed. A natural question arises — is such sacrifice (dropping *all* LO-criticality jobs) necessary whenever a HI-criticality job requires execution for more than its LO-WCET? The following example illustrates the potential benefits in terms of enhanced schedulability of the proposed probabilistic MC model.

Example 3.13. *Consider a system composed of the three independent MC implicit-deadline tasks that* $\tau_1 = \{[2,3], 0.1, 5, \text{HI}\}$, $\tau_2 = \{[3,4], 0.05, 10, \text{HI}\}$, *and* $\tau_3 = \{[1,1], 10, \text{LO}\}$, *to be scheduled on a preemptive uniprocessor, with desired system FPH threshold of* $F_S = 0.01$.

Since HI-*utilization of the system is* $u^H = 2/5 + 4/10 = 1$, *any deterministic MC scheduling algorithm will prioritize* τ_1 *and* τ_2 *over the* LO-*criticality task* τ_3, *and drop* τ_3 *if* any HI-*criticality job exceeds its* LO-*WCET.*

With the additional probability information provided in our richer model, however, more sophisticated scheduling and analysis can be done. Recall from the definition of f_i, τ_1 *has a probability of no larger than* 0.1 *to exceed a* 2-*unit execution within an hour, while the probability of any job in* τ_2 *exceeding a* 3-*unit execution within an hour is* 0.05. *Under the task-level independence assumption, the probability of jobs from both* HI-*criticality tasks requiring more than their* LO-*WCETs in an hour (*$P(x_1 = x_2 = 1) = P(x_1 = 1) \times P(x_2 = 1) = 0.1 \times 0.05 = 0.005$*) is smaller than* F_S[9]. *Hence, in the schedulability test of such system, we do not need to consider the case that* both HI-*criticality tasks exceed their* LO-*WCETs simultaneously.*

Moreover, either one of them exceeding its LO-*WCET will not result in an over-utilized system — a "server"* $\tau_s = \{0.2, 1, \text{HI}\}$ *can be added to provide the additional capacity (over and above the* LO-*WCET amount). This server will be scheduled and executed as a virtual task, and both* HI-*criticality tasks may run on the server.*

The total system utilization thus provisioned for the HI-*criticality tasks is* $2/5 + 3/10 + 0.2/1 = 0.9$; *upon provisioning an additional utilization of* $1/10 = 0.1$ *for the* LO-*criticality task* τ_3, *the*

[9]In general, we cannot simply ignore an event when its failure probability is below F_S. Instead, we do not need to consider a set of events only when the *sum* of their failure probability is below F_S. More details on this can be found in Sec. 3.2.4.

total utilization becomes 1*. Thus under any optimal uniprocessor scheduling strategy, e.g., EDF, the failure (any deadline miss) rate of the system in any hour will be no greater than F_S, and the MC instance is* strongly probabilistic schedulable *under this scheduling strategy (EDF plus the* HI*-criticality server) for the specified threshold F_S.*

3.2.4 Scheduling Strategy

3.2.4.1 The LFF-Clustering Algorithm

In this subsection, we present our strategy for scheduling independent preemptive MC task instances, by combining HI-criticality tasks into clusters intelligently, and provide a sufficient schedulability test for it. Consider what we have done in Example 3.13 above. We essentially: (i) conceptually combined the HI-criticality tasks τ_1 and τ_2 into a single cluster, provisioning an additional server into the system to accommodate their possible occasional HI-mode behaviors (execution beyond their LO-WCETs); and (ii) performed two EDF schedulability tests: one considering only HI-criticality tasks (with LO-WCETs) and the server, and the other also considering the LO-criticality task (τ_3). Since both tests succeed, we declare *strongly probabilistic schedulable* for the given instance; we would have declared *weakly probabilistic schedulable* if the second schedulability test had failed while the first one succeeded.

The technique that was illustrated in Example 3.13 forms the basis of the scheduling strategy that we derive in this section. To obtain a good upper bound to HI-criticality utilization of the system, we combine tasks into *clusters* — suppose that the n_{HI} HI-criticality tasks have been partitioned into M clusters $G_1, G_2, ..., G_M$, and let $y_i \in \{1, 2, ..., M\}$ denote to which cluster (number) task τ_i is assigned.

Definition 3.14 (Failure probability of a cluster)**.** Failure of a cluster G_m *is defined as job generated by more than one task in a single cluster exceeding their* LO*-WCETs within an hour. The probability*

of a failure occurring in cluster m is denoted as g_m and is given by

$$g_m \stackrel{def}{=} 1 - \prod_{i|y_i=m} (1 - f_i) - \sum_{j|y_j=m} f_j \frac{\prod_{i|y_i=m}(1 - f_i)}{1 - f_j}, \tag{3.4}$$

where the second term of right-hand side is the probability of no task (in the cluster) exceeding its LO-*WCET, and the last term represents the probability of exact one of the tasks exceeding its* LO-*WCET in an hour.*

Lemma 3.15. If $g_m < F_S/M$ holds for any cluster G_m, then the probability of having no failure in any cluster is greater than $(1 - F_S)$.

Proof: Since clusters do not overlap with each other (each HI-criticality task belongs to a single cluster) and thus are independent to each other, the probability of having no failure in any cluster is given by the product of each cluster being failure-free, which is: $\prod_{m=1}^{M}(1 - g_m) > \prod_{m=1}^{M}(1 - F_S/M) = (1 - F_S/M)^M \geq 1 - F_S$ (From Binomial Theorem). □

Lemma 3.15 provides a safe *failure threshold* F_S/M for each cluster; i.e., the rule for forming *clusters* is $g_m < F_S/M$, where M is the current number of clusters.

The *additional utilization cost of a cluster* G_m is defined to be equal to the additional utilization cost (δ_i) of the task within the cluster with the largest δ_i value; i.e.,

$$\Delta_m \stackrel{\text{def}}{=} \max_{i|\tau_i \in G_m} \delta_i. \tag{3.5}$$

The *total system additional utilization cost* is given by the sum of additional utilization cost of all M clusters;

$$\Delta \stackrel{\text{def}}{=} \sum_{m=1}^{M} \Delta_m. \tag{3.6}$$

A critical observation is that, if a task τ_i with additional utilization cost δ_i has been assigned to a cluster, assigning any other task τ_j with $\delta_j \leq \delta_i$ to the cluster will not increase the additional utilization cost. To minimize the total additional utilization cost of the entire task set, we therefore greedily expand existing clusters with tasks of larger additional utilization cost while ensuring that

the relationship $g_m < F_S/M$ continues to hold, leading to the Largest Fit First (LFF)-Clustering algorithm.

Algorithm 1: Algorithm LFF-Clustering

```
Input: F_S, {f_i}_{i=1}^{n_HI}, {δ_i}_{i=1}^{n_HI}
Output: maximum total additional utilization cost Δ
begin
    Sort the tasks in non-increasing order of δ_i;
    m ← 1, M ← n_HI, y_i ← 0 for i = 1,...,n;
    while ∏_{i=1}^{n_HI} y_i = 0 (an unassigned task exists) do
        Δ_m ← 0 (additional utilization of each cluster);
        for i ← 1 to n_HI do
            if y_i > 0:  continue;
            y_i ← m, M ← M − 1;
            if g_m ≥ F_S/M:  y_i ← 0, M ← M + 1;
        Δ_m ← max_{i|y_i=m} δ_i;
        m ← m + 1;
    return ∑_{m=1}^{M} Δ_M;
```

This algorithm greedily expands each existing cluster with unassigned tasks while the condition $g_m < F_S/M$ holds; while a new cluster is created only if it is not possible to assign a task to any current cluster without violating the condition $(g_m < F_S/M)$.

Remark 1. Similar to what has been done in (Díaz et al., 2002) and (Maxim and Cucu-Grosjean, 2013), we may achieve a precise distribution to the total utilization of all tasks by applying the *convolution* operation '$\otimes$', which results in an exponential ($O(2^{n_{\text{HI}}})$, to be precise) running time (see Sec. 3.2.4.2). The sufficient schedulability test based on the LFF-Clustering algorithm runs in $O(n_{\text{HI}}^2)$ time, where n_{HI} is the number of HI-criticality tasks.

Remark 2. In the case that *all tasks share the same f_i value*, the schedulability test based on LFF-Clustering becomes *necessary and sufficient*.

Run-Time Strategy. During execution, a HI-criticality *server* τ_s with utilization Δ and a period of 1 tick is added to the task system. We need the server period as 1 tick because the mechanism

and the analysis will not work if there is release or deadline within a server period. At any time instant that the server is executing, the active[10] HI-criticality job, if any, with the earliest deadline, is executed; if there is no such job, the current job of the server is dropped[11]. All jobs including the server are scheduled and executed in EDF order, and a job is dropped at its deadline if it is not completed by then.

Note that although we introduce a server task with a period of 1, preemption does not necessarily happen that often. The goal of the sever task with utilization Δ is to preserve a "bandwidth" of at least Δ for HI- criticality jobs if the HI-criticality ready queue is not empty. There are three situations to be considered:

Situation 1: The job with the earliest deadline is a HI-criticality job. In this situation, we execute the HI-criticality job with 100% processor share, and no more preemption is incurred by the server.

Situation 2: The job with the earliest deadline is a LO-criticality job *and* the HI-criticality ready queue is empty. In this situation, we execute the LO-criticality job with 100% processor share, and hence, there is no additional preemption in this situation either.

Situation 3: The job with the earliest deadline is a LO-criticality job *and* the HI-criticality ready queue is *not* empty. In this situation, we want to preserve a processor share of Δ for HI- criticality jobs and to execute the LO-criticality ones with the rest $1-\Delta$ of the processor capacity. Therefore, the server creates preemptions every time unit.

That is, only in Situation 3, our algorithm "introduces" extra preemptions due to the server scheme, and normal EDF scheduling is applied in other cases. One may claim that such server allocation scheme may results in more preemptions than the approaches where the server capacity is only used for overruns. Actually, this is because that the goal here is trying *not to drop* LO-criticality tasks even when a few HI-criticality ones exceed their LO-WCETs. Thus, in order to guarantee HI-deadline being met always, we have to make certain use of the server even when no HI-criticality

[10]A job is *active* if it is released and incomplete at that time instant.

[11]Since an integer model of time is assumed (i.e., all task periods are integers and all job arrivals occur at integer instants in time), and the server has a period of 1, it is safe to drop the current job of the server if there is no active HI-criticality job since there can be no HI-criticality job releases in the current period of the server.

behavior is detected — simply taking "precautions". The alternative way such as assigning HI-criticality jobs virtual deadlines may lead to fewer preemptions, at a cost of losing the performance of schedulability ratio (see experimental comparisons).

In this work, we make use of servers to implement our algorithms and prove the possibility of proficiently apply failure probability to both MC modeling and MC scheduling. In future work, we will release the server period assumption of 1 unit of time by applying adaptivity to resource reservation (Santinelli et al., 2011; Stoimenov et al., 2010). With the analysis of the deadline and task periods, we will be able to implement realistic servers which adapt their period and budget to the MC-scheduler needs, while leaving the system predictable at any time interval. Such adaptive behavior will not introduce any overhead, and mostly will allow not to miss task deadline.

3.2.4.2 The Convolution Based Approach

There are two HI-criticality tasks in Example 3.13. As a result, only one combinatorial event needs to be eliminated, which is $x_1 = x_2 = 1$; i.e., jobs of both tasks exceed their WCETs in an hour of execution. When the number of HI-criticality tasks (n_{HI}) becomes larger, there will be more indicator variables (x_i), and we need to calculate the probability of all $2^{n_{\text{HI}}}$ combinations in order to achieve an exact analysis. Similar to what's been done in (Díaz et al., 2002) and (Maxim and Cucu-Grosjean, 2013), the sum of two random variables $\mathscr{X}$ and $\mathscr{Y}$ is defined as the *convolution* $\mathscr{X} \otimes \mathscr{Y}$ where $P(\mathscr{Z} = z) = \sum_k P(\mathscr{X} = k)P(\mathscr{Y} = z-k)$, in case of discrete random variables.

We associate the probabilities with the possible utilization values using the following notation for the utilization pdf:

$$f_{\mathscr{U}_i} = \begin{pmatrix} u_i^L = \frac{C_i^L}{T_i} & u_i^H = \frac{C_i^H}{T_i} = u_i^L + \delta_i \\ f_i & 1 - f_i \end{pmatrix}, \tag{3.7}$$

and calculate the exact total utilization of multiple tasks by applying the convolution operation. For example, the convolution for the utilizations of two HI-criticality tasks in Example 3.13 is:

$$\begin{pmatrix} 0.4 & 0.6 \\ 0.9 & 0.1 \end{pmatrix} \otimes \begin{pmatrix} 0.3 & 0.4 \\ 0.95 & 0.05 \end{pmatrix} = \begin{pmatrix} 0.7 & 0.8 & 0.9 & 1.0 \\ 0.855 & 0.045 & 0.095 & 0.005 \end{pmatrix}.$$

By applying the $\otimes$ operation (for $(n_{\text{HI}} - 1)$ times) to all HI-criticality tasks, we will end up with a capacity requirement distribution, consisting of as many as $2^{n_{\text{HI}}}$ rows. According to the system failure threshold F_S, we could easily determine the maximum HI-criticality capacity *needed to be considered* from such capacity distribution by the definition of probabilistic schedulability, and ignore the rest *highly unlikely* executions.

Considering the instance in Example 3.13, we can only ignore the 1.0-HI-utilization case if $F_S = 0.01$. However, if we require weaker confidence level to the system failure probability, e.g., $F_S = 0.1$, both the 0.9- and the 1.0-HI-utilization cases can be ignored since the probability of τ_1 and τ_2 requiring a total utilization of more than 0.8 is $0.095 + 0.005 \leq F_S$. As a result, the system will pass probabilistic schedulability test even LO-criticality utilization is up to 0.2.

Such precise calculation by a chain of $\otimes$ operations requires exponential time ($O(2^{n_{\text{HI}}})$, to be precise). Some work on distribution re-sampling; i.e., (Maxim et al., 2012), makes the distribution of convolution less complex by using pessimistic distributions with fewer values (keeping the worst values for safety issues). The trade-off is between complexity and accuracy.

3.2.4.3 Schedulability Test

It is evident that for *strongly probabilistic schedulable* (i.e., to ensure that the probability of missing *any* deadline is no larger than the specified system FPH F_S — see Definition 3.11), it is (necessary and) sufficient that $\left(\sum_{i=1}^{n} C_i^L / T_i + \Delta\right)$ must be no larger than the capacity of the processor (which is 1).

For *weakly probabilistic schedulable* (i.e., to ensure that the probability of missing any HI-*criticality* deadline is no larger than F_S — see Def. 3.11), it is necessary that $\left(\sum_{i|\chi_i=\text{HI}} C_i^L/T_i + \Delta\right)$ must be no larger than 1 as well. The following theorem helps establish a sufficient condition for ensuring weakly probabilistic schedulable:

Theorem 3.16. If no job exceeds its LO-WCET, then no deadline is missed if

$$\Delta \cdot (1 - \sum_{i|\chi_i=\text{HI}} \frac{C_i^L}{T_i}) + \sum_{i=1}^{n} \frac{C_i^L}{T_i} \leq 1. \tag{3.8}$$

Proof: As assumed, the task set is feasible when no job exceeds its LO-WCET; i.e., $\sum_{i=1}^{n} C_i^L/T_i \leq 1$. Therefore, if the server does not exist, all task will meet their deadlines under EDF scheduling. Since the server task is not a real task but only executes the earliest-deadline HI-criticality job if exists, introducing this server will never *delay* any HI-criticality task's execution (comparing to no-server circumstance). Thus, the deadlines of all HI-criticality jobs will still be met.

Next, by contradiction, we show if (3.8) holds, all deadlines of LO-criticality jobs will also met. Suppose t_d is the first time instant when a deadline of a LO-criticality job is missed. Let t_0 denote the last idle instant for jobs with deadlines at or before t_d[12], then $[t_0, t_d)$ is a busy interval. Let Ψ denote the set of the HI-criticality jobs that are released at or after t_0 and with deadlines at or before t_d, and Ψ' denote the complement (i.e., HI-criticality jobs with deadlines after t_d).

Let W denote the total demand created by jobs in Ψ within $[t_0, t_d)$, then

$$W \leq \sum_{i|\chi_i=\text{HI}} \left\lfloor \frac{t_d - t_0}{T_i} \right\rfloor \cdot C_i^L. \tag{3.9}$$

We have shown that all HI-criticality jobs will meet their deadlines (in the first paragraph of this proof), which implies that there must be a processor supply of W allocated to those jobs in Ψ. Since the server has a period of 1, no job will be released during each server period. Moreover, the server has the highest scheduling priority, and will execute the earliest-deadline HI-criticality job (when

[12]If at an instant there is no active job with a deadline at or before t_d, it is considered *idle* in this proof.

exists). Thus for any unit-length period, if jobs in Ψ are executed for a cumulative length of w, at least a server budget of $\Delta \cdot w$ will be consumed by those jobs. Thus, within $[t_0, t_d)$, at least $\Delta \cdot W$ server budget must execute jobs in Ψ. On the other hand, the server (by its definition) could have at most $\Delta \cdot (t_d - t_0)$ budget in $[t_0, t_d)$. Thus, within the period $[t_0, t_d)$, jobs in Ψ' will consume server budget of at most $\Delta \cdot (t_d - t_0) - \Delta \cdot W$. Moreover, since there will always be active jobs with deadline at or before t_0 throughout the interval, and we are using pure EDF "outside" the server, jobs in Ψ' (with later deadlines) can only execute within $[t_0, t_d)$ by consuming server budget.

Also, within the busy interval $[t_0, t_d)$, a LO-criticality task τ_i can only release $\lfloor (t_d - t_0)/T_i \rfloor$ jobs with deadlines at or before t_d. Thus, and by the definition of t_d and t_0, we have

$$\big(\Delta \cdot (t_d - t_0) - \Delta \cdot W\big) + W + \sum_{i|\chi_i = \mathrm{LO}} \left\lfloor \frac{t_d - t_0}{T_i} \right\rfloor \cdot C_i^L > t_d - t_0. \tag{3.10}$$

Moreover,

$$\begin{aligned}
& \big(\Delta \cdot (t_d - t_0) - \Delta \cdot W\big) + W + \sum_{i|\chi_i = \mathrm{LO}} \left\lfloor \frac{t_d - t_0}{T_i} \right\rfloor \cdot C_i^L \\
= \ & \Delta \cdot (t_d - t_0) + (1 - \Delta) \cdot W + \sum_{i|\chi_i = \mathrm{LO}} \left\lfloor \frac{t_d - t_0}{T_i} \right\rfloor \cdot C_i^L \\
\leq \ & \{\text{by (3.9) and } \Delta \leq 1\} \\
& \Delta \cdot (t_d - t_0) + (1 - \Delta) \cdot \sum_{i|\chi_i = \mathrm{HI}} \left\lfloor \frac{t_d - t_0}{T_i} \right\rfloor \cdot C_i^L + \sum_{i|\chi_i = \mathrm{LO}} \left\lfloor \frac{t_d - t_0}{T_i} \right\rfloor \cdot C_i^L \\
\leq \ & \{\text{by } \lfloor (t_d - t_0)/T_i \rfloor \leq (t_d - t_0)/T_i \text{ for all } i \text{ and } \Delta \leq 1\} \\
& \Delta \cdot (t_d - t_0) + (1 - \Delta) \cdot \sum_{i|\chi_i = \mathrm{HI}} \frac{t_d - t_0}{T_i} \cdot C_i^L + \sum_{i|\chi_i = \mathrm{LO}} \frac{t_d - t_0}{T_i} \cdot C_i^L \\
= \ & \Delta \cdot (t_d - t_0) - \Delta \cdot (t_d - t_0) \cdot \sum_{i|\chi_i = \mathrm{HI}} \frac{C_i^L}{T_i} + \\
& (t_d - t_0) \cdot \sum_{i|\chi_i = \mathrm{HI}} \frac{C_i^L}{T_i} + (t_d - t_0) \cdot \sum_{i|\chi_i = \mathrm{LO}} \frac{C_i^L}{T_i} \\
= \ & \Delta \cdot (t_d - t_0) \cdot \left(1 - \sum_{i|\chi_i = \mathrm{HI}} \frac{C_i^L}{T_i}\right) + (t_d - t_0) \cdot \sum_{i=1}^{n} \frac{C_i^L}{T_i}.
\end{aligned} \tag{3.11}$$

By (3.10) and (3.11),

$$\Delta \cdot (t_d - t_0) \cdot \left(1 - \sum_{i|\chi_i=\text{HI}} \frac{C_i^L}{T_i}\right) + (t_d - t_0) \cdot \sum_{i=1}^{n} \frac{C_i^L}{T_i} > t_d - t_0, \tag{3.12}$$

Canceling $(t_d - t_0)$ on both sides contradicts our theorem assumption, (3.8).

Thus, such t_d does not exist and therefore no LO-criticality job will miss its deadline. □

Theorem 3.16 yields the schedulability test pMC (Algorithm 2), while Theorem 3.17 below establishes its correctness.

Theorem 3.17. The schedulability test pMC is *sufficient* in the following sense:

- If it returns *strongly probabilistic schedulable*, the probability of any task missing its deadline is no greater than F_S; and
- if it returns *weakly probabilistic schedulable*, the probability of any HI-criticality task missing its deadline is no greater than F_S, and no deadline is missed when all jobs finish upon execution of their LO-WCETs.

Proof: From Lemma 3.15 and Theorem 3.16, we may conclude that the possibility of HI-criticality tasks altogether requiring an additional utilization of no more than Δ is less than F_S, and thus they can still meet their deadlines with probability no less than $1 - F_S$ upon the assigned server task.

The utilization-based test of EDF is run twice. If the first test succeeds; i.e., total utilization (including the server) is less than 1, then all tasks will meet their deadlines with a probability no less than $(1 - F_S)$ — this ensures strongly probabilistic schedulable. If not, we need to check two other conditions which together ensure *weakly probabilistic schedulable*: (i) a utilization test involving HI-criticality tasks and the server, which guarantees that the probability of all HI-criticality tasks meeting their deadlines is no less than $(1 - F_S)$ should some jobs exceed their LO-WCETs; and (ii) a utilization based condition involving all tasks and the server, which guarantees *correctness* for all tasks when no HI-criticality one exceeds its LO-WCET (Theorem 3.16).

□

The schedulability test pMC returns *strongly probabilistic schedulable* if we are able to schedule the system such that the probability of missing any deadline is at most the specified threshold F_S, or *weakly probabilistic schedulable* if we are able to schedule the system such that the probability of missing any HI-criticality deadline is at most F_S. We will then use EDF to schedule and execute the task set with LO-WCETs and the additional server task $\tau_s = \{\Delta, 1, \text{HI}\}$.

In the case that the schedulability test pMC returns *unknown*, we are not able to schedule the system using the proposed probabilistic analysis technique. Normally it is either we have set a too high safety requirement to the system; i.e., the threshold F_S is too small, or the WCET estimations are not precise enough for HI-criticality tasks; i.e., the f_i's are not small enough comparing to F_S (and n_{HI}), and/ or the C_i^L's are still not differentiable enough with respect to C_i^H's.

We show how our algorithm works by applying it to an example.

Example 3.18. *Consider the MC task system consisting of six tasks shown in Table 3.3, and a specified allowed system FPH of $F_S = 3.2 \times 10^{-4}$. For simplicity, tasks are ordered decreasingly by δ_i values. (The δ_i's for each task are calculated according to* (3.3).*)*

Table 3.3: A set of MC tasks.

-	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6
$[C_i^L, C_i^H]$	$[0.5, 0.5]$	$[1, 3]$	$[1, 2]$	$[3, 5]$	$[3, 6]$	$[1, 2]$
T_i	5	10	25	50	20	4
δ_i	0.2	0.2	0.08	0.06	0.05	-
f_i	0.0001	0.0001	0.01	0.01	0.001	-
χ_i	HI	HI	HI	HI	HI	LO

The LFF-Clustering algorithm initially assigns each task a single cluster, and try to expand the one (of the largest Δ_i value) with task τ_1. τ_2 can be combined into Cluster G_1 since $g_1 < F_S/M$ holds ($g_1 = 1-(1-f_1)(1-f_2) - f_1(1-f_2) - f_2(1-f_1) = f_1 f_2 < F_S/4$). Similarly, combining τ_3 will result in a smaller number of total remaining clusters ($M = 3$), and Inequality $g_1 < F_S/M$ continues to hold.

However, this inequality no longer holds as we further expand G_1 for τ_4 (g_1 becomes greater than $F_S/2$). Thus, we assign Task τ_4 a second cluster G_2. Similar to the situation of τ_4, τ_5 cannot be combined into cluster G_1. However, combining τ_5 with τ_4 is allowed since $M = 2$ and $g_2 = f_4 f_5 < F_S/2$.

Finally, we have visited all HI*-criticality tasks, and the value to be returned by the LFF-Clustering algorithm is $\Delta_1 + \Delta_2 = u_1 + u_4 = 0.26$.*

Since the total system utilization (including the LO*-criticality task τ_6) remains less than 1 with a server of utilization* 0.26. *The schedulability test pMC returns* strongly probabilistic schedulable. *During run-time, an additional server $\tau_s = \{0.26, 1, \text{HI}\}$ will be added to the task system, on which active* HI*-criticality jobs will execute (also in EDF order). When there is no active* HI*-criticality job, the current job of the server will be dropped.*

Algorithm 2: Schedulability Test pMC

Input: τ, F_S
Output: schedulability

begin
  Calculate the δ_i values for all HI-criticality tasks in τ;
  $u^L \leftarrow \sum_{i=1}^{n} C_i^L / T_i$;
  $u_H^L \leftarrow \sum_{i|\chi_i=\text{HI}} C_i^L / T_i$;
  $\Delta \leftarrow$ *LFF-Clustering*(F_S, $\{f_i\}_{i=1}^{n_{\text{HI}}}$, $\{\delta_i\}_{i=1}^{n_{\text{HI}}}$);
  if $u^L + \Delta \leq 1$ **then**
    return *strongly probabilistic schedulable;*
  else
    if $u_H^L + \Delta \leq 1$, $\Delta \cdot (1 - u_H^L) + u^L \leq 1$ **then**
      return *weakly probabilistic schedulable;*
  return *unknown;*

3.2.5 Schedulability Experiments

We have conducted schedulability tests on randomly-generated task systems, comparing our proposed method with existing one. The objective was to demonstrate the benefits of our model: by adding a probability estimation f_i to each task, our algorithm may successfully schedule (return *probabilistically correct* or *partial probabilistically correct*) many task sets that are unschedulable according to existing MC-scheduling algorithms; e.g., the EDF-VD algorithm (Baruah et al., 2012b).

Since this is the first work that combines pWCET and schedulability with mixed-criticality, it's hard to find a proper baseline to compare with. The reason EDF-VD is selected here since (i) it is a widely accepted MC scheduling strategy, (ii) it is the most general algorithm in the whole VD family, and (iii) HI-criticality tasks are treated as a whole in both algorithms — EDF-VD sets virtual deadline according to a common factor, while we make use of a HI-criticality server.

3.2.5.1 MC Task Generator

Our MC task generator results from a minor modification of the workload-generation algorithm introduced by Guan et al. (Guan et al., 2013). The input parameters for our workload generation algorithm are as follows:

1. U_{bound}: The desired value of the larger of LO-criticality and HI-criticality utilization of the task system: $\max\left(U_L^L(\tau)+U_H^L(\tau), U_H^H(\tau)\right)$.

2. $[U_L, U_U]$: Utilizations are uniformly drawn from this range; $0 \leq U_L \leq U_U \leq 1$ ($[0,1]$ as default).

3. $[T_L, T_U]$: Task periods are uniformly drawn from this range; $0 < T_L \leq T_U$. Note that many proposed schedulability tests are utilization based, and thus periods play no role in the experiments.

4. $[Z_L, Z_U]$: The ratio (or fudge factor) of the HI-criticality utilization of a task to its LO-criticality utilization is uniformly drawn from this range; $1 \leq Z_L \leq Z_U$.

5. P: The probability that a task is a HI-criticality task; $0 \leq P \leq 1$ (0.5 as default).

To generate a task system for a given combination of parameter values, the task-generation algorithm repeatedly adds tasks to an initially empty system until the utilization bound is met (see (Guan et al., 2013)).

This workload generator has passed an Artifact Evaluation[13] process. The Matlab code of the generator is available at:

`http://www.cs.unc.edu/~zsguo/files/MC_task_creator.zip`.

3.2.5.2 Schedulability Comparison

For the experiments in this section, the parameter u^L is ranged from 0 to 1, while u^H is ranged from 0 to 1.5, each with step 0.01. Each task set contains 20 tasks, each of which is assigned LO or HI criticality with equal probability. LO-criticality utilizations are assigned according to *UUniFast*; given an expected HI utilization u^H, we inflate the LO-criticality utilizations of the HI-criticality tasks using random factors chosen to ensure that the cumulative HI utilization of the task-set equals the desired value with high probability.

Among the $626,200$ valid task sets that we generated, EDF-VD succeeds to schedule $306,299$ (48.9%) of them, and the proposed pMC reports probabilistic schedulable for a total of 438787 sets (70.1%), and only $121,426$ sets (19.4%) are reported unknown. Even when focused only upon systems for which HI-criticality utilization is less than 1, EDF-VD fails to schedule 18.0%, while pMC returns unknown for only 8.4% of the sets. Figure 3.2 depicts the schedulability results for the two algorithms, where $f_i = 10^{-3}$ of all tasks τ_i and $F_s = 10^{-6}$. Instances with similar u^L and u^H values are put into a same small block. The color of each small block represents the percentage of schedulable sets[14]. As shown in Figure 3.2, although EDF-VD and pMC do not dominate each other, pMC generally significantly outperforms EDF-VD, particularly upon task-sets with large HI-utilization.

[13] For additional details, please refer to http://ecrts.org/artifactevaluation.

[14] Since we randomly assign criticality levels to all tasks, the LO utilization of HI-criticality tasks is expected to be $u^L/2$. It is unlikely to generate tasks with $u^H < u^L/2$, and thus the right lower triangle regions are left blank in Figure 3.2.

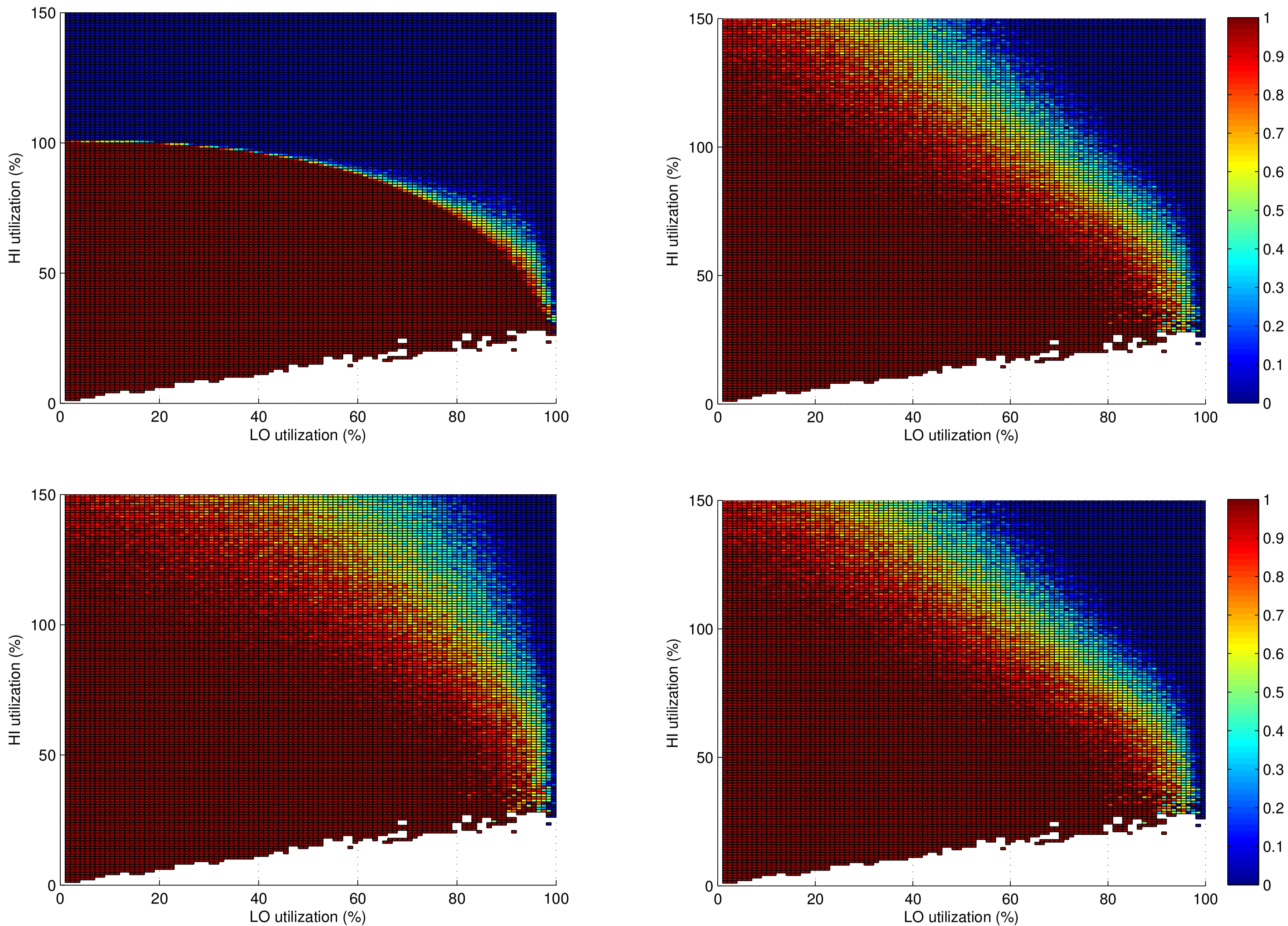


Figure 3.2: Schedulability comparison of EDF-VD (upper-left), pMC when returning partial correct or correct (lower-left), and pMC returning only correct (right, duplicated for easy comparison), where the color of each block represents the percentage of schedulable sets within certain *utilization* ranges. (*Please enlarge these figures enough, and/or use colored printer for better view.*)

To show the robustness of our algorithm with respect to different f_i distributions, we focus on task sets with HI utilization between 0.9 and 1. Figure 3.3 reports the ratios of schedulable (i.e., weakly probabilistic schedulable) sets over different LO utilizations. With the additional probability information, the schedulable ratio is significantly improved for heavy tasks comparing to EDF-VD (Baruah et al., 2012b).

The introduced parameter f_i is assigned to tasks in different ways; i.e., all sharing the same value, following the uniform distribution, or following the log-uniform distribution ($f_i = 10^x$, where x is uniformly chosen). Generally speaking, smaller average f leads to a higher ratio of acceptance, and there is no significant difference between different distributions of f_i with the same average, which indicates that our algorithm is *robust* to different combinations of output measurement probabilities from probabilistic timing analysis tools.

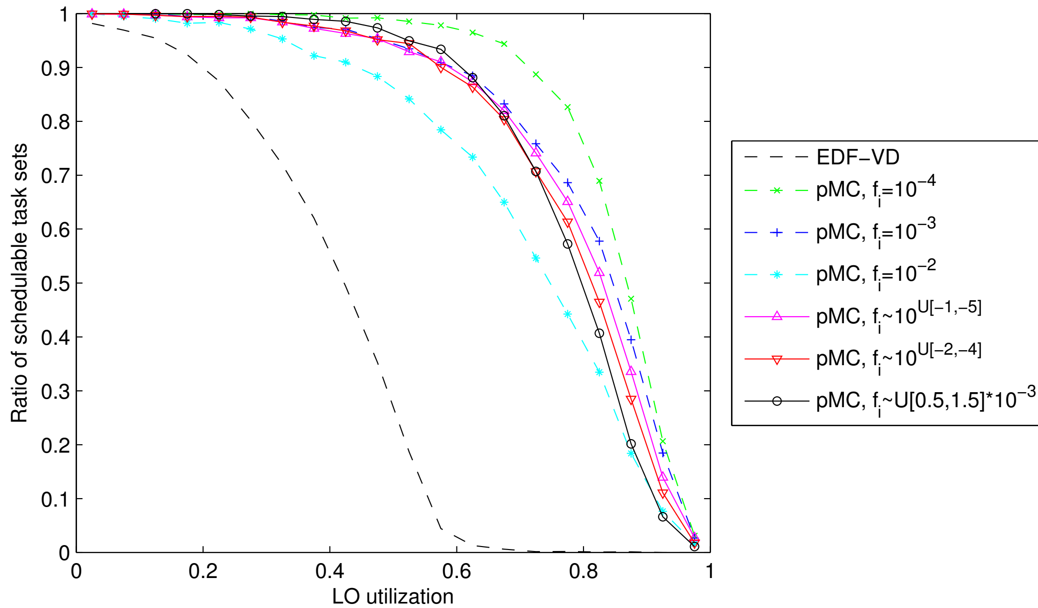


Figure 3.3: Schedulability ratio comparison of EDF-VD and pMC, where HI utilization varies from 0.9 to 1 in a uniform manner.

3.3 MC Task Scheduling on Multi-Processor

It has been shown that optimal speedup of $4/3$ can be achieved by EDF-VD when scheduling MC tasks on a uniprocessor (Li and Baruah, 2010). However, when facing multiprocessors, the best-known speedup factor among all schedulers has been $\sqrt{5}+1 \approx 3.24$ until late 2014, by Global-EDF-VD (Li and Baruah, 2012). Very recently, the speedup cost is improved to $(1+\sqrt{5})/2 \approx 1.618$ by a fluid based algorithm named MC-Fluid (Lee et al., 2014). Since fluid schedules are not always implementable upon actual computing platforms, another algorithm, named MC-DP-Fair, was also derived in (Lee et al., 2014) that transforms such a fluid schedule into a schedule in which each task is assigned either zero or one processor at each instant in time.

In this section, for scheduling implicit-deadline dual-criticality task sets, we further prove that the speedup of MC-Fluid is $4/3$, and show its optimality such that no algorithm can achieve a smaller speedup bound. We propose a much simpler scheduler named *MCF*, with only utilization based schedulability test and the same speedup factor as MC-Fluid. Note that MC-DP-Fair continues to be valid for use in conjunction with Algorithm MCF; hence, we will not address the issue of constructing non-fluid schedules any further. Instead, we will assume that the schedule constructed by Algorithm MCF is passed on to MC-DP-Fair to be converted into a non-fluid schedule, just as the schedules constructed by MC-Fluid were in (Lee et al., 2014). Most of the contributions made in this section can be found at (Baruah et al., 2015).

3.3.1 System Model and Prior Work

The MC-Fluid scheduling algorithm (Lee et al., 2014) was designed for scheduling mixed-criticality implicit-deadline sporadic task systems upon identical multiprocessor platforms. Given such a task system, MC-Fluid determines a scheduling strategy under the *fluid scheduling* model (see Sec. 2.3.3). This allows for schedules in which individual tasks may be assigned a fraction ≤ 1 of a processor (rather than an entire processor, or none) at each instant in time, subject to the

constraint that the sum of the fractions assigned to all the tasks does not exceed the sum of the computing capacities of all the processors at any instant.

System Model. Let τ denote a collection of n dual-criticality implicit-deadline sporadic tasks that are to be scheduled upon m unit-speed processors. A task τ_i is characterized by the parameters $(C_i^L, C_i^H, T_i, \chi_i)$, where $\chi_i \in \{\text{LO}, \text{HI}\}$ denotes its criticality, C_i^L and C_i^H its LO and HI criticality WCETs ($C_i^L \leq C_i^H$), and T_i its period. As a general rule, $\tau_H \subseteq \tau$ ($\tau_L \subseteq \tau$) denotes all the HI-criticality tasks (LO-criticality tasks, respectively) in τ. We adapt the utilization notations described in (2.7).

System behavior. Similar to the cases considered in previous sections of this chapter, the system may execute at two different modes: it runs at the LO-criticality mode until some HI-criticality job (of a task) has been executed for C_i^L time units and does not signal its finishing.

Objective. The objective is to schedule the task set τ upon m unit-speed processors in an *MC correct* manner (which is similar to the correctness definition for MC jobs, yet is described formally in detail here for the sake of completeness):

Definition 3.19 (MC-correct for tasks)**.** *A scheduling strategy is MC-correct if it ensures that*

- *During any execution of the system in which each job of each task completes upon executing for no more than the task's* LO*-criticality WCET, all jobs complete by their deadlines; and*

- *during any execution of the system in which each job of each task completes upon executing for no more than the task's* HI*-criticality WCET, all jobs of all the* HI*-criticality tasks complete by their deadlines (while jobs of* LO*-criticality tasks may fail to do so).*

MC-Fluid. To do so, MC-Fluid seeks to determine per-mode *execution rates* θ_i^L and θ_i^H for each task τ_i such that the scheduling algorithm depicted in Figure 3.4 constitutes an MC-correct scheduling strategy for τ.

3.3.2 Algorithm MCF

In this section, we describe Algorithm MCF for scheduling dual-criticality implicit-deadline sporadic task set τ upon m identical processors. MCF follows the fluid schedule framework (like

- Each τ_i initially executes at a constant rate θ_i^L. That is, at each time instant it is executing upon θ_i^L fraction of a processor (here, θ_i^L is required to be ≤ 1).
- If a job of any task τ_i does not complete despite having received C_i^L units of execution (equivalently, having executed for a duration (C_i^L/θ_i^L)), then
  - All LO-criticality tasks are immediately discarded, and
  - Each HI-criticality task henceforth executes at a constant rate θ_i^H (θ_i^H, too, must be ≤ 1).

Figure 3.4: The run-time scheduling strategy used by Algorithm MC-Fluid

MC-Fluid) and seeks to find the proper execution rates θ_i^L and θ_i^H such that MC correctness is guaranteed by the run-time algorithm depicted in Figure 3.4. The manners in which Algorithm MCF computes these θ_i^L, θ_i^H values are depicted in Figure 3.5; the steps are explained below.

Observe that $(U_L^L + U_H^L)$ denotes the total system utilization in LO-criticality behaviors, and U_H^H the total system utilization in HI-criticality behaviors. Hence, for τ to be feasible on a platform of m unit-speed processors, it is necessary that $\left(U_L^L + U_H^L\right) \leq m$, $U_H^H \leq m$ and $u_i^H \leq 1$ for each $\tau_i \in \tau_H$. The value assigned to ρ (Expression (3.13)) should therefore be ≤ 1 for any feasible system. Informally speaking, the quantity $(1-\rho)$ can be thought of as representing the "slack" or excess capacity in the system; we seek to exploit this slack by setting the execution rates (the θ_i^L's and θ_i^H's) to be greater than the utilizations (the u_i's).

If ρ is indeed ≤ 1, then the execution rates at HI-criticality (the θ_i^H's) for the HI-criticality tasks are set equal to their HI-criticality utilizations u_i^H scaled by a factor $1/\rho$ (Expression (3.14)). The execution rates at LO-criticality (the θ_i^L's) for each LO-criticality task is set equal to the task utilization (u_i^L), while the θ_i^L for each HI-criticality task is set according to the formula given in Expression (3.15). The correctness of these assignments will be formally proved in Sec. 3.3.3 below.

Finally, the assignment of execution rates is declared a success if the θ_i^L values that are assigned sum to no more than the cumulative computing capacity of the platform.

1. Define ρ as follows:

$$\rho \leftarrow \max\Big\{\Big(\frac{U_L^L + U_H^L}{m}\Big), \Big(\frac{U_H^H}{m}\Big), \max_{\tau_i \in \tau_H}\{u_i^H\}\Big\} \tag{3.13}$$

2. **If** $\rho > 1$ **then** declare failure; **else** assign values to the execution rate variables as follows:

$$\theta_i^H \quad \leftarrow \quad u_i^H/\rho \text{ for all } \tau_i \in \tau_H \tag{3.14}$$

$$\theta_i^L \quad \leftarrow \quad \begin{cases} \frac{u_i^L \theta_i^H}{\theta_i^H - \left(u_i^H - u_i^L\right)}, & \text{if } \tau_i \in \tau_H \\ u_i^L, & \text{else (i.e., if } \tau_i \in \tau_L) \end{cases} \tag{3.15}$$

3. **If**

$$\sum_{\tau_i \in \tau} \theta_i^L \leq m \tag{3.16}$$

then declare success **else** declare failure

Figure 3.5: Algorithm MCF

We now illustrate the manner in which Algorithm MCF computes the θ_i^L and θ_i^H parameters via a simple example.

Example 3.20. *Consider the dual-criticality implicit-deadline sporadic task system that in Table 3.4, which is to be scheduled upon a 2-processor platform.*

	T_i	C_i^L	C_i^H	χ_i	u_i^L	u_i^H
τ_1	10	3	8	HI	0.3	0.8
τ_2	20	8	14	HI	0.4	0.7
τ_3	30	3	3	HI	0.1	0.1
τ_4	40	20	20	LO	0.5	0.5

Table 3.4: Example task system

For this task system,

$$\begin{aligned}\rho &= \max\left\{\frac{.3+.4+.1+.5}{2}, \frac{.8+.7+.1}{2}, \max\{.8,.7,.1\}\right\}\\ &= \max\{1.3/2, 1.6/2, .8\}\\ &= 0.8\end{aligned}$$

Therefore tasks τ_1, τ_2 and τ_3, get θ_i^H values assigned as follows:

$$\begin{aligned}\theta_1^H &= \frac{0.8}{0.8} = 1.0\\ \theta_2^H &= \frac{0.7}{0.8} = 0.875\\ \textit{and } \theta_3^H &= \frac{0.1}{0.8} = 0.125\end{aligned}$$

The assigned θ_i^L values are as follows:

$$\begin{aligned}\theta_1^L &= \frac{1.0\times 0.3}{1.0-(0.8-0.3)} = 0.6\\ \theta_2^L &= \frac{0.875\times 0.4}{0.875-(0.7-0.4)} = \frac{14}{23} < 0.61\\ \theta_3^L &= \frac{0.125\times 0.1}{0.125-(0.1-0.1)} = 0.1\\ \textit{and } \theta_4^L &= 0.5\end{aligned}$$

Since

$$\sum_{i=1}^{4}\theta_i^L < \left(0.6+0.61+0.1+0.5\right) = 1.81,$$

we conclude that the task system is indeed schedulable by Algorithm MCF.

Run-time complexity. Algorithm MCF has run-time that is *linear* in the number of tasks in τ (i.e., $\Theta(n)$): the scaling factor ρ can be computed in one pass through the task system; the θ_i^H and θ_i^L values in a second pass; and checking the sum of θ_i^L values does not exceed m in a third pass.

3.3.3 Correctness of MCF

We now prove that the proposed MCF is correct: if Algorithm MCF computes the execution rates without declaring failure for a given task system τ, then the schedule resulting from using these execution rates in the manner described in Figure 3.4 does indeed constitute an MC-correct scheduling strategy.

Lemma 3.21. *Assigned execution rates (θ_i^H and θ_i^L) are all* $\leq \mathbf{1}$.

Proof: Observe that $\rho \geq u_i^H$ for all $\tau_i \in \tau$. It follows that $\theta_i^H = (u_i^H/\rho)$ is always ≤ 1, as required.

With regards to the θ_i^L's, the value assigned to θ_i^L for each LO-criticality task is equal to u_i^L (and hence ≤ 1). For high criticality tasks, by Equation (3.15), θ_i^L for each $\tau_i \in \tau_H$ is assigned a value $\frac{u_i^L \theta_i^H}{\theta_i^H - \left(u_i^H - u_i^L\right)}$. This is $\leq \theta_i^H$ if

$$\begin{aligned} & \frac{u_i^L}{\theta_i^H - \left(u_i^H - u_i^L\right)} \leq 1 \\ \Leftrightarrow \quad & u_i^L \leq \theta_i^H - \left(u_i^H - u_i^L\right) \\ \Leftrightarrow \quad & u_i^H \leq \theta_i^H \end{aligned}$$

which follows from the requirement that ρ be ≤ 1 (else, we would have declared failure).

As a result, for each HI-criticality task, we have

$$\theta_i^H \geq \theta_i^L. \tag{3.17}$$

I.e., the execution rate guaranteed to each HI-criticality task does not *decrease* upon identification of HI-criticality behavior), and thus, θ_i^L variables are also assigned values ≤ 1. □

Condition (3.16) ensures that the assignment of values to the θ_i^L variables does not exceed the capacity of the *m*-processor platform; Lemma 3.22 below shows that neither does the assignment of values to the θ_i^H variables.

Lemma 3.22.

$$\sum_{\tau_i \in \tau_H} \theta_i^H \leq m \tag{3.18}$$

Proof: It follows from Equation (3.13) that

$$\begin{aligned} & \rho \geq \frac{U_H^H}{m} \\ \Leftrightarrow \quad & \frac{U_H^H}{\rho} \leq m \end{aligned} \tag{3.19}$$

We use this inequality to conclude that

$$\left(\sum_{\tau_i \in \tau_H} \theta_i^H\right) = \left(\sum_{\tau_i \in \tau_H} \frac{u_i^H}{\rho}\right) = \left(\frac{1}{\rho} \sum_{\tau_i \in \tau_H} u_i^H\right) = \left(\frac{U_H^H}{\rho}\right) \leq m$$

and Condition (3.18) is shown to hold. □

Lemma 3.23 below asserts that the execution rate assigned to each task in a steady LO-criticality or HI-criticality behavior is adequate. Mode transition part will be considered later.

Lemma 3.23. *For each $\tau_i \in \tau$*

$$\theta_i^L \geq u_i^L; \tag{3.20}$$

$$\theta_i^H = \left(\frac{u_i^H}{\rho}\right) \geq u_i^H. \tag{3.21}$$

Proof: This is clearly true for each $\tau_i \in \tau_L$, since $\theta_i^L = u_i^L$ for all such τ_i. To see that it is also true for each $\tau_i \in \tau_H$, observe that for each such τ_i,

$$\begin{aligned} \theta_i^L &= u_i^L \times \frac{\theta_i^H}{\theta_i^H - \left(u_i^H - u_i^L\right)} \\ &\geq u_i^L \ \ (\text{Since } \left(u_i^H - u_i^L\right) \geq 0) \end{aligned}$$

Since $\rho \leq 1$, it is obvious that $\theta_i^H \geq u_i^H$. □

Finally, we show that the θ-values computed by Algorithm MCF ensure MC-correctness in HI-criticality behaviors during a mode transition, by analyzing the point in time during run-time at which it is detected that some job has executed beyond its LO-criticality WCET.

Lemma 3.24. *Let t_o denote the first time instant at which some job does not signal completion despite having executed for its* LO*-criticality WCET. Any* HI*-criticality job that is active (i.e., that has been released but has not completed execution) at time instant t_o receives an amount of execution no smaller than its* HI*-criticality WCET prior to its deadline.*

Proof: Suppose that a job of HI-criticality task τ_i is active at time instant t_o. Let us suppose that it had arrived at time instant $(t_o - w)$, where w is a positive number $\leq T_i$; its deadline is then at time instant $(t_o - w + T_i)$. Over the interval $[t_o - w, t_o)$, this job will have received an amount of execution equal to $\theta_i^L \times w$; since the job is still active, it must be the case that

$$\begin{aligned} & \theta_i^L \times w \leq C_i^L \\ \Leftrightarrow \quad & w \leq \frac{C_i^L}{\theta_i^L} \end{aligned} \tag{3.22}$$

From the instant t_o to its deadline — i.e., over the interval $[t_o, t_o - w + T_i)$, of duration $(T_i - w)$ — the job of τ_i will execute at a rate θ_i^H. Hence for this job to meet its deadline, it is sufficient that

$$\begin{aligned} & w\theta_i^L + (T_i - w)\theta_i^H \geq C_i^H \\ \Leftrightarrow \quad & T_i\theta_i^H - w(\theta_i^H - \theta_i^L) \geq C_i^H \\ \Leftarrow \quad & T_i\theta_i^H - \frac{C_i^L}{\theta_i^L}(\theta_i^H - \theta_i^L) \geq C_i^H \text{ (By Inequality (3.22))} \\ \Leftrightarrow \quad & \theta_i^H - \frac{u_i^L}{\theta_i^L}(\theta_i^H - \theta_i^L) \geq u_i^H \\ \Leftrightarrow \quad & \theta_i^H - \frac{u_i^L\theta_i^H}{\theta_i^L} + u_i^L \geq u_i^H \\ \Leftrightarrow \quad & \theta_i^H \geq (u_i^H - u_i^L) + \frac{u_i^L\theta_i^H}{\theta_i^L} \\ \Leftrightarrow \quad & 1 \geq \frac{u_i^H - u_i^L}{\theta_i^H} + \frac{u_i^L}{\theta_i^L} \end{aligned} \tag{3.23}$$

By Equation (3.15), for each $\tau_i \in \tau_H$ we have

$$\begin{aligned} & \theta_i^L = \frac{u_i^L \theta_i^H}{\theta_i^H - \left(u_i^H - u_i^L\right)} \\ \Leftrightarrow \quad & \frac{\theta_i^H - \left(u_i^H - u_i^L\right)}{\theta_i^H} = \frac{u_i^L}{\theta_i^L} \\ \Leftrightarrow \quad & 1 - \left(\frac{\left(u_i^H - u_i^L\right)}{\theta_i^H}\right) = \frac{u_i^L}{\theta_i^L} \\ \Leftrightarrow \quad & \frac{u_i^L}{\theta_i^L} + \frac{u_i^H - u_i^L}{\theta_i^H} = 1 \end{aligned}$$

thereby establishing Condition (3.23) and completing the proof of the lemma. □

Theorem 3.25. *Values assigned to the θ_i^H and θ_i^L variables according to Equations* (3.14)-(3.15) *that satisfy Condition* (3.16) *constitute an MC-correct schedule.*

Proof: Lemma 3.23 and Condition (3.16) together suffice to establish correctness in any LO-criticality behavior. Similarly, Lemmas 3.21 and 3.22 establishes correctness in "steady state" following the transition to HI-criticality behavior. And finally, Lemma 3.24 establishes that MC-correctness is also preserved upon a transition from LO-criticality to HI-criticality behavior. □

3.3.4 Speedup of MCF and MC-Fluid

Prior to our work, the best-known speedup for multiprocessor MC scheduler is $(1+\sqrt{5})/2$ or approximately 1.618 for MC-Fluid, shown in (Lee et al., 2014). We prove a better (i.e., lower) speedup bound of $4/3$ for Algorithm MCF in this subsection.

Lemma 3.26. *Let c denote any positive constant. The function*

$$f(x) = \frac{x(c-x)}{\frac{c}{3}+x}$$

is $\leq \frac{c}{3}$ for all values of $x \in [0,c]$.

Global maximum:

$$\max\left\{\frac{x(1-x)}{\frac{1}{3}+x} \,\middle|\, 0 \le x \le 1\right\} = \frac{1}{3} \text{ at } x = \frac{1}{3}$$

Plot:

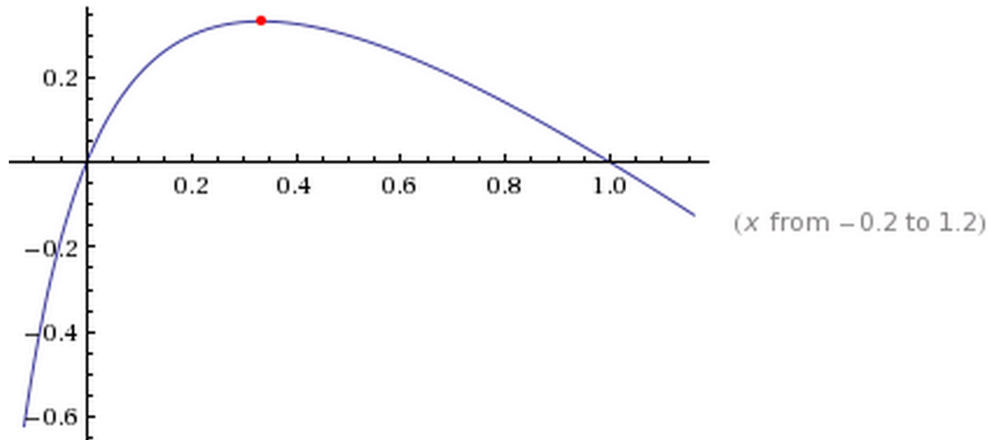


Figure 3.6: Plot of $f(x)$ for $c = 1$ (made with the WolframAlpha® computational knowledge engine: `https://www.wolframalpha.com/`)

Proof: This lemma is easily proved rigorously using standard techniques from the calculus: taking the derivative of $f(x)$ with respect to x, we see that the only value of $x \in [0, c]$ where this derivative equals zero is $x \leftarrow c/3$. We therefore conclude that $f(x)$ takes on its maximum value over $[0, c]$ for some $x \in \{0, c/3, c\}$. Explicit computation of $f(x)$ at each of these values reveals that the value is maximized at $x = c/3$, where it takes on the value $c/3$. (We skip the details of the derivation here; for a visual depiction of $f(x)$, it is plotted as a function of x in Figure 3.6.) □

Theorem 3.27. *Algorithm MCF is speedup-optimal for scheduling dual-criticality implicit-deadline task systems: it has a speedup factor of* $4/3$*, and no non-clairvoyant algorithm may have a speedup factor lower than* $4/3$.

Proof: We first show the speedup factor of $4/3$; i.e., under condition $\rho \leq 3/4$, the θ_i^H, θ_i^L values computed by Algorithm MCF (in the manner specified in Expressions (3.14)–(3.15) of Figure 3.5) satisfy Condition (3.16).

Let us first rewrite Condition (3.16) to an equivalent form expressed in Condition (3.24) below.

$$
\begin{aligned}
&\sum_{\tau_i \in \tau} \theta_i^L \leq m \\
&\Leftrightarrow \quad \sum_{\tau_i \in \tau_L} \theta_i^L + \sum_{\tau_i \in \tau_H} \theta_i^L \leq m \\
&\Leftrightarrow \quad U_L^L + \sum_{\tau_i \in \tau_H} \frac{u_i^L \theta_i^H}{\theta_i^H - \left(u_i^H - u_i^L\right)} \leq m \\
&\Leftrightarrow \quad U_L^L + \sum_{\tau_i \in \tau_H} u_i^L \Big(1 + \frac{u_i^H - u_i^L}{\theta_i^H - \left(u_i^H - u_i^L\right)}\Big) \leq m \\
&\Leftrightarrow \quad U_L^L + \sum_{\tau_i \in \tau_H} u_i^L + \sum_{\tau_i \in \tau_H} \frac{u_i^L (u_i^H - u_i^L)}{\theta_i^H - \left(u_i^H - u_i^L\right)} \leq m \\
&\Leftrightarrow \quad U_L^L + U_H^L + \sum_{\tau_i \in \tau_H} \frac{u_i^L \left(u_i^H - u_i^L\right)}{\theta_i^H - \left(u_i^H - u_i^L\right)} \leq m
\end{aligned}
\tag{3.24}
$$

We will show, in the remainder of this proof, that if $\rho \leq 3/4$ then Condition (3.24) is satisfied; this will serve to establish the correctness of Lemma 3.27.

Let us assume henceforth that $\rho \leq 3/4$. From the definition of ρ (Expression (3.13)), it follows that

$$
U_L^L + U_H^L \leq \frac{3}{4} m \tag{3.25}
$$

$$
U_H^H \leq \frac{3}{4} m \tag{3.26}
$$

$$
\forall \tau_i \in \tau_H \ \ u_i^H \leq \frac{3}{4} \tag{3.27}
$$

Additionally, since $\theta_i^H \leftarrow u_i^H / \rho$, it must hold that

$$
\forall \tau_i \in \tau_H \ \ \theta_i^H \geq \frac{4}{3} u_i^H \tag{3.28}
$$

Let us use Inequalities (3.25)–(3.28) to further simplify Condition (3.24).

$$
\begin{aligned}
& U_L^L + U_H^L + \sum_{\tau_i \in \tau_H} \frac{u_i^L\left(u_i^H - u_i^L\right)}{\theta_i^H - \left(u_i^H - u_i^L\right)} \leq m \\
\Leftarrow \quad & \frac{3}{4}m + \sum_{\tau_i \in \tau_H} \frac{u_i^L\left(u_i^H - u_i^L\right)}{\theta_i^H - \left(u_i^H - u_i^L\right)} \leq m \text{ (By Ineq. (3.25))} \\
\Leftarrow \quad & \frac{3}{4}m + \sum_{\tau_i \in \tau_H} \frac{u_i^L\left(u_i^H - u_i^L\right)}{\frac{4}{3}u_i^H - \left(u_i^H - u_i^L\right)} \leq m \text{ (By Ineq. (3.28))} \\
\Leftrightarrow \quad & \frac{3}{4}m + \sum_{\tau_i \in \tau_H} \frac{u_i^L\left(u_i^H - u_i^L\right)}{\frac{u_i^H}{3} + u_i^L} \leq m \\
\Leftrightarrow \quad & \sum_{\tau_i \in \tau_H} \frac{u_i^L\left(u_i^H - u_i^L\right)}{\frac{u_i^H}{3} + u_i^L} \leq \frac{m}{4} \\
\Leftarrow \quad & \sum_{\tau_i \in \tau_H} \frac{u_i^H}{3} \leq \frac{m}{4} \quad \text{(By Lemma 3.26)} \\
\Leftrightarrow \quad & \frac{1}{3}U_H^H \leq \frac{m}{4} \\
\Leftarrow \quad & \frac{1}{3} \times \frac{3}{4}m \leq \frac{m}{4} \quad \text{(By Inequality (3.26))} \\
\Leftrightarrow \quad & \frac{m}{4} \leq \frac{m}{4}
\end{aligned}
$$

It has previously been shown (Baruah et al., 2012b, Theorem 5) that no non-clairvoyant algorithm for scheduling dual-criticality implicit-deadline sporadic task systems can have a speedup factor smaller than $4/3$ even on *uniprocessors* (i.e., for $m = 1$), which is a special case for multiprocessor. Thus, a smaller speedup is not possible for *any* non-clairvoyant algorithm, and the $4/3$ speedup is optimal. □

It was shown in (Lee et al., 2014) that Algorithm MC-Fluid has a speedup bound no worse than $(1+\sqrt{5})/2$ (≈ 1.618) for dual-criticality implicit-deadline sporadic task systems. We will now improve this result and show that MC-Fluid, like MCF, has a speedup bound no worse than $4/3$.

Corollary 3.28. *The speedup factor of MC-Fluid is* $4/3$.

Proof. The result comes from the domination relationship between MC-Fluid and MCF: If Algorithm MCF (Figure 3.5) computes θ_i^H and θ_i^L values for a given dual-criticality implicit-deadline

sporadic task system τ without declaring failure, then the θ_i^H values so computed satisfy Inequalities (3.29)–(3.31) (and τ is therefore successfully scheduled by MC-Fluid as well).

Let us suppose that Algorithm MCF (Figure 3.5) computes θ_i^H and θ_i^L values for a given dual-criticality implicit-deadline sporadic task system τ without declaring failure. Since ρ, as computed by Expression (3.13) of Figure 3.5, must be ≤ 1, it follows that $\theta_i^H = u_i^H/\rho$ is $\leq u_i^H$ and Inequality (3.29) is satisfied for all $\tau_i \in \tau_H$.

It is shown (Lee et al., 2014, Theorem 2) that the convex optimization problem solved by MC-Fluid essentially computes θ_i^H values to satisfy the following inequalities:

$$\begin{aligned} \forall i: \tau_i \in \tau_H: \; u_i^H &\leq \theta_i^H & (3.29)\\ U_L^L + U_H^L + \sum_{\tau_i \in \tau_H} \frac{u_i^L\left(u_i^H - u_i^L\right)}{\theta_i^H - \left(u_i^H - u_i^L\right)} &\leq m & (3.30)\\ \sum_{\tau_i \in \tau_H} \theta_i^H &\leq m & (3.31) \end{aligned}$$

Since $\rho \geq U_H^H/m$, it follows that

$$\begin{aligned} &\rho \geq \frac{U_H^H}{m}\\ &\Leftrightarrow \quad \rho \geq \frac{\sum_{\tau_i \in \tau_H} u_i^H}{m}\\ &\Leftrightarrow \quad m \geq \frac{\sum_{\tau_i \in \tau_H} u_i^H}{\rho}\\ &\Leftrightarrow \quad m \geq \sum_{\tau_i \in \tau_H} \frac{u_i^H}{\rho}\\ &\Leftrightarrow \quad m \geq \sum_{\tau_i \in \tau_H} \theta_i^H \end{aligned}$$

and Inequality (3.31) is also satisfied.

It remains to show that Inequality (3.30) is satisfied as well. Observe that

$$U_L^L + U_H^L + \sum_{\tau_i \in \tau_H} \frac{u_i^L\left(u_i^H - u_i^L\right)}{\theta_i^H - \left(u_i^H - u_i^L\right)}$$

$$
\begin{aligned}
&= U_L^L + \sum_{\tau_i \in \tau_H} \Big(u_i^L + \frac{u_i^L(u_i^H - u_i^L)}{\theta_i^H - (u_i^H - u_i^L)} \Big) \\
&= U_L^L + \sum_{\tau_i \in \tau_H} \Big(\frac{u_i^L \theta_i^H - u_i^L(u_i^H - u_i^L) + u_i^L(u_i^H - u_i^L)}{\theta_i^H - (u_i^H - u_i^L)} \Big) \\
&= U_L^L + \sum_{\tau_i \in \tau_H} \Big(\frac{u_i^L \theta_i^H}{\theta_i^H - (u_i^H - u_i^L)} \Big) \\
&= U_L^L + \sum_{\tau_i \in \tau_H} \theta_i^L \\
&= \sum_{\tau_i \in \tau_L} \theta_i^L + \sum_{\tau_i \in \tau_H} \theta_i^L \\
&= \sum_{\tau_i \in \tau} \theta_i^L
\end{aligned}
$$

which is indeed $\leq m$, according to Inequality (3.16).

Thus, we show that any task system that is successfully scheduled by Algorithm MCF is also successfully scheduled by MC-Fluid, and the $4/3$ speedup immediately yields from Theorem 3.27. □

Remark. It has been shown in (Lee et al., 2014) that MC-Fluid is an optimal execution rate assignment algorithm; i.e., if a set of θ_i^L's and θ_i^H's exist for a specified dual-criticality implicit-deadline sporadic task system that constitutes an MC-correct fluid scheduling strategy, then MC-Fluid is guaranteed to find at least one such assignment. The above lemma directly follows from this result as well.

Note. Experimental comparisons on MCF, MC-Fluid, and existing MC schedulers were conducted by our collaborators. From the experimental study, it is shown that MCF outperforms global fpEDF (Bini and Buttazzo, 2005), global fixed-priority (Audsley, 2001), and partitioned EDF (Baruah et al., 2012b) by a considerable margin, for all the task set parameter combinations. The performance gap continues to widen for increasing number of processors. The performance of MCF and MC-Fluid is relatively similar, and primarily depends on the normalized utilization bound $U_B = \max\{(U_L^L + U_H^L)/m, U_H^H/m\}$. For more details on those experiments, please refer to (Baruah et al., 2015).

3.4 Summary

In this chapter, we focus on Vestal's interpretation of MC scheduling, where MC solely arises from WCET estimations. Under this model, multiple WCET thresholds will be assigned to a single piece of code, and its run-time behavior remains unknown. In a dual-criticality system, any HI-criticality job may trigger a mode switch of the whole system when it exhausted its LO-criticality WCET (which is less pessimistic) and does not signal finishing. The correctness of the system consists of separate validations under each running mode. More precisely, deadline meeting guarantees are made to all tasks under LO-criticality mode, while only to more important ones under HI-criticality mode.

Although many nice scheduling theory results exist since Vestal's pioneering work (Vestal, 2007), we have shown that improvements to existing schedulers can be made, e.g.,

- Via proposing new schedulers. Sec. 3.1 proposes Algorithm LE-EDF for scheduling MC job set. It is shown that it is computationally more efficient than OCBP, while strictly dominate it (i.e., any job set that is schedulable by OCBP is schedulable by LE-EDF, but not the other way around). The experimental study also suggests LE-EDF out-performs a more complicated and recently proposed algorithm named MCEDF.

- Via proposing new MC workload models. Sec. 3.2 adds one more parameter into the Vestal model, which captures the probability information about the uncertainties. By assuming independences, we remove Vestal's assumption about all tasks *simultaneously* violating their LO-criticality WCETs. Experiments on randomly generated task sets suggest that some simple uniprocessor MC algorithms may outperform state-of-the-art ones and use computing resources much more efficiently.

- Via providing better analytical results to existing schedulers. Sec. 3.3 provides an improved speedup result (from $(\sqrt{5}+1)/2$ to $4/3$) for an existing algorithm named MC-Fluid for MC task scheduling upon multiprocessor platform. This result is optimal in the sense that no

lower speedup is possible for the problem (due to its NP-hardness under non-clairvoyance). This closes a 5-year long open problem and helps us understand how efficient MC-Fluid is.

CHAPTER 4: WHEN MC ARISES FROM VARYING-SPEED PLATFORMS

As MC systems increasingly come to be implemented on commodity processors, we believe that it is imperative that real-time scheduling theory understands how to implement these systems to meet the twin goals of providing correctness guarantees at high levels of assurance to the more critical functionalities while simultaneously making efficient use of platform resources.

In this chapter, we seek to study the scheduling of MC systems upon CPUs which may be modeled as varying-speed processors. As pointed out in Sec. 1.1.2, the uncertainty of estimations arises from the executing speed of the platform as well. In order to make correctness guarantees at very high levels of assurance, it may be necessary to consider the possibility that the processor is executing at a very low speed.

A natural question arises regarding the usefulness of bringing in new models: "Why not use longer WCETs to model slower executing platforms?" The central thesis (see Sec. 1.2) gives a direct answer to this: existing scheduling methods may be adapted at no significant capacity loss in some cases, while in some other cases new mechanisms can be developed, with better performance. This answer will be supported by the results described in Secs. 4.2 – 4.5 as we consider various kinds of combinations of workload and platforms. To begin with, we formally describe our interpretation of the MC system where MC arises from the variations of executing speeds in Sec. 4.1.

4.1 Our MC Interpretation: The Varying-Speed Platform MC Model

In this chapter, we take a novel perspective on mixed-criticality scheduling: the mixed-criticality nature of the system arises in the fact that while we would like all functionalities to execute correctly

under normal circumstances of the platform, it is essential that the more critical functionalities execute correctly even under (unlikely) pathological conditions of the processor(s).

To express this formally, we model the workload of an MC system as normal real-time workloads with a criticality level $\chi_i \in \{1, 2, \ldots, m\}$ expressing its degree of importance (where larger values indicate greater importance). We desire to schedule the system upon a platform with varying-speed processor(s). Each processor is characterized by a sequence of m speeds $1 = s_1 > s_2 > \ldots > s_m$: under normal circumstances it completes at least one unit of execution during each time unit (equivalently, it executes as a speed-1, or faster, processor), it may at any instant fall into "degraded" modes. Under degraded mode at level $l \in 2, ..., m$, the processor can complete fewer than one, but at least s_l, units of execution during each time unit. Similar to the settings in Sec 3.1.1, it is not a priori known when, or whether, such degradation will occur (non-clairvoyant).

Objective. In general, we seek a scheduling strategy that for each criticality level $l, 1 \leq l \leq m$ *guarantees to correctly execute all those jobs that have criticality* $\geq l$*, provided the processor speed(s) never falls below* s_l *during run-time.*

Remark. Some systems are capable of *self-monitoring*: it immediately knows if and when such degradation occurs; i.e., it has access to some facility similar to the capabilities offered by the Linux `cpufreq-info` command, while some may not. Both self-awareness properties will be considered in this chapter.

The following example illustrates this model.

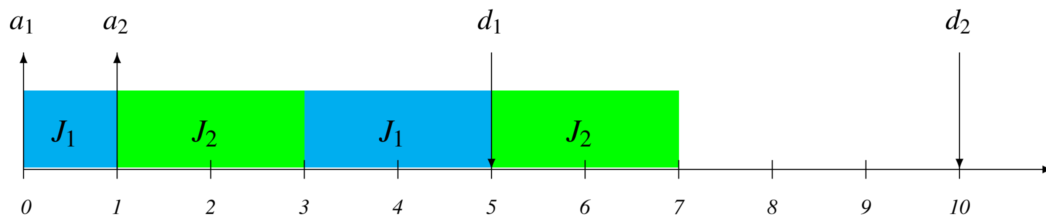
Example 4.1. *Consider the following collection of two jobs, to be scheduled on a preemptive processor with specified speeds* $s_1 = 1$ *and* $s_2 = \frac{1}{2}$*:*

Job	*Criticality*	*Release date*	*WCET*	*Deadline*
J_1	LO	*0*	*3*	*5*
J_2	HI	*1*	*4*	*10*

An Earliest Deadline First (EDF) (Liu and Layland, 1973) schedule for this system prioritizes J_1 *over* J_2*. This is fine if the processor does not degrade:* J_1 *executes over the interval* $[0, 3)$ *and* J_2 *over* $[3, 7)$*, thereby resulting in both deadlines being met.*

Now suppose that the processor was to degrade at some instant within the time-interval $[0,10]$: a correct scheduling strategy should execute the HI*-criticality job J_2 to complete by its deadline (although it may fail to execute J_1 correctly). But consider the scenario where the processor degrades to some speed $s' < \frac{4}{7}$, or ≈ 0.55) starting at time instant 3: in the EDF schedule, J_2 would obtain merely $(10-3) \times s' < 4$ units of execution prior to its deadline at time instant 10. We therefore conclude that EDF does not schedule this system correctly.*

An alternative scheduling strategy could instead execute jobs as follows on a normal (speed-1) processor: J_1 over the interval $[0,1)$; J_2 over $[1,3)$; J_1 again, over $[3,5)$; and finally J_2 over $[5,7)$:



If the processor degrades to a speed < 1 at any instant during this execution then J_1 is immediately discarded and the processor executes J_2 exclusively.

It may be verified that this scheduling strategy will result in J_2 completing by its deadline regardless of when (if at all) the processor degrades to any speed ≥ 0.5, and in both deadlines being met if the processor remains normal (or degrades at any instant ≥ 5).

Note. Although in this chapter we have chosen to model the problem in terms of real-time jobs executing on varying-speed processors, the model (and our results) are also applicable to the transmission of time-sensitive data on potentially bandwidth-varying communication media. Specifically, they are particularly relevant to data communication problems in which time-sensitive data and data streams must be transmitted over communications media which can provide a high bandwidth under most circumstances but can only *guarantee* some lower bandwidths: the high bandwidth would correspond to the normal processor speed and the lower bandwidths to the degraded speeds. We therefore believe that the work described in this chapter is relevant to problems of factory communication, communication within automobiles or aircraft, wireless sensor networks, etc., in addition to processor scheduling of mixed-criticality workloads.

4.2 MC Job Scheduling on Self-Monitored Uniprocessor

A Self-Monitored Uniprocessor a processor is characterized by several execution speeds: a normal speed and several levels of degraded speeds. Under normal circumstances it will execute at or above its normal speed; conditions during run-time may cause it to execute slower. It is desired that all components of the MC workload execute correctly under normal circumstances. If the processor speed degrades, it should nevertheless remain the case that the more critical components execute correctly (although the less critical ones need not do so).

In this section, we derive a linear program (LP) based *optimal* algorithm for scheduling MC workloads upon such platforms. We do not restrict the total number of criticality levels in the system. However, we do assume that the system is capable of self-monitoring: it immediately knows if and when a degradation occurs. The optimality result shows the privilege of our MC model (separately characterizing uncertainties in platform execution speed), as achieving optimality in schedulability under the Vestal model has shown (Baruah, 2016)(Baruah et al., 2012a) to be highly computationally intractable (NP-hard in the strong sense).

We first formally describe the system model and discuss the relationship to prior work based on Vestal's interpretation in Sec. 4.2.1, then give our algorithm (TDMC-LP) in Sec. 4.2.2, and show its properties in Sec. 4.2.3. Finally, for the only time in this dissertation, 4.2.5 studies non-preemptive scheduling by showing the NP-hardness for MC job scheduling even under the varying-speed platform model. Most of the contributions made in this section can be found at (Baruah and Guo, 2013) and (Guo and Baruah, 2014a).

4.2.1 Model and Relationship to Prior Work

We start out considering MC systems that can be modeled as collections of *independent jobs*. Each mixed-criticality (MC) job J_i is characterized by a 4-tuple of parameters: a release date a_i, a WCET c_i, a deadline d_i, and a criticality level $\chi_i \in \{1, 2, \ldots, m\}$. Note that this WCET c_i is

measured based upon some constant unit-speed processor — a job with WCET of c_i may require a period of length c_i/s when executing on a speed-s processor, for some $s < 1$.

Let $t_1, t_2, \ldots, t_{k+1}$ denote the at most $2n$ distinct values for the release date and deadline parameters of the n jobs, in increasing order (i.e., $t_j < t_{j+1}$ for all j). These release dates and deadlines partition the time-interval $\left[\min_i\{a_i\}, \max_i\{d_i\}\right)$ into k *intervals*, which we will denote as $I_1, I_2, \ldots, I_k$, with I_j denoting the interval $[t_j, t_{j+1})$.

A mixed-criticality *instance* I is specified by specifying

- a finite collection of MC jobs $J = \{J_1, J_2, \ldots, J_n\}$, and
- a varying-speed processor that is characterized by a normal speed s_1 (without loss of generality, assumed to be 1) and some specified *degraded processor speeds* $s_2, \ldots, s_m$ in strictly decreasing order; i.e., $s_m < s_{m-1} < \ldots < s_2 < 1$.

The interpretation is that the jobs in J are to execute on a single shared processor that has m modes: a *normal* mode and $(m-1)$ *degraded* modes. In the normal mode, the processor executes as a unit-speed processor and hence completes one unit of execution per unit time, whereas in degraded mode l it completes fewer than s_{l-1}, but at least s_l, units of execution per unit time, for $l = 2, \ldots, m$.

The processor starts out executing at its normal speed. It is not *a priori* known when, if at all, the processor will degrade: this information only becomes revealed during run-time when the processor actually begins executing at a slower speed. We seek to determine a ***correct scheduling strategy*** which is formally defined as follows:

Definition 4.2 (correct scheduling strategy)**.** *A scheduling strategy for MC instances is* correct *if it possesses the property that upon scheduling any MC instance $I = (J = \{J_1, J_2, \ldots, J_n\}, s_1, \ldots, s_m)$, each job J_i completes by its deadline if the processor executes at a speed $\geq s_{\chi_i}$ throughout its scheduling window $[a_i, d_i)$.*

Much of prior research considers a model in which each job is characterized by multiple WCETs — the results can indeed be directly applied to our problem: Consider a job in our setting that has WCET C and is being scheduled on a varying-speed processor with normal speed $s_1 = 1$

and degraded speeds $s_2, \ldots, s_m$. This job may be represented in the multiple-WCET model as a job with a WCET vector of $C, C/s_2, \ldots, C/s_m$. If all jobs execute for no more than their normal WCETs, then all jobs should execute correctly; while if some jobs execute beyond their normal WCETs, then only some of the jobs (those with criticality levels exceeding a particular value) are required to execute correctly.

It is not difficult to show that the algorithms proposed in prior work for scheduling Vestal's MC systems (with multiple WCET specifications) can be used to schedule this transformed system, and that the resulting scheduling strategy correctly schedules the MC system under our interpretation (upon the varying-speed processor). Hence, all the problems considered in this section could in principle be solved by simply transforming to the earlier, multiple-WCET, model, and applying the previously-proposed solution techniques.

However, in a latter part of this section, we show that one can sometimes do *better* than such an approach. It was observed to be so because the problem we are considering here, of MC scheduling on varying-speed processors, is *simpler* (from a computational complexity perspective) than the previously-considered problem of MC scheduling with multiple-WCETs specified. For instance, whereas determining preemptive uniprocessor feasibility for a collection of independent MC jobs specified according to the multiple-WCET model is known (Baruah et al., 2012a) to be NP-hard in the strong sense, in Sec. 4.2.2 we will present an optimal polynomial-time algorithm for solving the same problem in our model. For the case of dual-criticality systems of implicit-deadline sporadic tasks on preemptive uniprocessors, a speedup lower bound of $4/3$ had been established (Baruah et al., 2012b) for the multiple-WCETs model, whereas we will also provide an optimal (speedup-1) algorithm.

4.2.2 Algorithm TDMC-LP

In this subsection, we present an efficient strategy for scheduling preemptable mixed-criticality job set. We start out with a general **overview** of our strategy. Given an instance I, prior to run-time we will construct a scheduling table $S(I)$, which prescribes the amounts of execution for the

specified intervals. During run-time, scheduling decisions are made according to this scheduling table. Amounts within each interval are executed in the priority order of their criticality levels (more important first), and we just perform the best-effort execution over all assigned amounts. A job is dropped at its deadline if it is not finished. Note that we do not discard a job with criticality level lower than ℓ even when the processing speed has (been detected) fallen to some value in the range $(s_{\ell+1}, s_\ell]$ — such mechanism improves the chances of lower criticality jobs meeting their deadlines even when the processor degrades severely[1].

In the remainder of this subsection, we present a simple linear-programming based algorithm for constructing the scheduling table $S(I)$ optimally. By *optimal*, we mean that if there is a correct scheduling strategy (Definition 4.2 above) for an instance I, then the scheduling strategy described above is a correct with the scheduling table we will construct. Its properties including correctness and optimality will be provided in Sec. 4.2.3.

We start out identifying the following (obvious) necessary condition for MC-schedulability:

Lemma 4.3. *In order that a correct scheduling strategy exists for MC instance $I = (J, s_1, \ldots, s_m)$, it is* necessary *that for each criticality level $l = 1, \ldots, m$, EDF correctly schedules all the jobs in I with criticality level $\geq l$ upon a speed-s_l uniprocessor.* □

Given any instance I, it can be efficiently determined whether I satisfies the necessary conditions of Lemma 4.3: for each l, simply simulate the EDF scheduling of all the jobs in I with criticality-level $\geq l$ upon a speed-s_l processor. In the remainder of this section, let us therefore assume that any instance under consideration satisfies these necessary conditions. (I.e., any instance that fails these conditions can obviously not have a correct scheduling strategy, and is therefore flagged as being unschedulable.)

Given an MC instance $I = (\{J_1, J_2, \ldots, J_n\}, s_1, \ldots, s_m)$ that satisfies the conditions of Lemma 4.3, we now describe how to construct a linear program (LP) such that a feasible solution for this linear program can be used to construct scheduling table $S(I)$.

[1] An example of such benefit will be shown in the execution analysis (Item 2) of Example 4.4, where J_2 with criticality level of 2 may meet its deadline in the case that the processer falls into a slowest functional speed of s_3 since $t = 1$.

To construct our linear program we define $n \times k$ variables $x_{i,j}$, $1 \leq i \leq n; 1 \leq j \leq k$. Variable $x_{i,j}$ denotes the amount of execution we will assign to job J_i in the interval I_j, in the scheduling table that we are seeking to build.

The following n constraints specify that each job receives adequate execution in the normal schedule:

$$\Big(\sum_{j \mid t_j \geq a_i \wedge d_i \geq t_{j+1}} x_{i,j}\Big) \geq c_i, \text{ for each } i, 1 \leq i \leq n; \tag{4.1}$$

while the following k constraints specify the capacity constraints of the intervals:

$$\big(\sum_{i=1}^{n} x_{i,j}\big) \leq s_1(t_{j+1} - t_j), \text{ for each } j, 1 \leq j \leq k. \tag{4.2}$$

Within each interval, the scheduling table will execute jobs in the priority order of their criticality levels; i.e., amounts from higher criticality level jobs get executed first. (That is, the interval I_j will have a block of level-m criticality execution of duration $\sum_{i:\chi_i=m} x_{i,j}$, followed by blocks of l-criticality execution of duration $\sum_{i:\chi_i=l} x_{i,j}$ with l from $m-1$ down to 1, in order.) It should be evident that any scheduling table generated in this manner from $x_{i,j}$ values satisfying the above $(n+k)$ constraints will execute all jobs to completion upon a normal (non-degraded) processor. It now remains to write constraints for specifying the requirements with respect to degraded conditions — that the higher criticality jobs complete execution even in the event of the processor degrading into corresponding modes.

Since within each interval, amounts are executed according to the priority of criticality level, we observe that the worst-case scenarios occur when the processing speed drops at the very *beginning* of a time interval, since that would leave the minimum computing capacity. All amounts will be executed at the best effort, and a job can only be dropped at its deadline when unfinished by then. For each $\{p, l\}$, $1 \leq p \leq k, 2 \leq l \leq m$, we represent the possibility that the processor degrades into speed-s_l mode at the start of the interval I_p in the following manner:

1. Suppose that the processor degrades into speed-s_l mode at time instant t_p; i.e., the start of the interval I_p. Henceforth, only jobs of criticality $\geq l$ must be fully executed in order to meet their deadlines.

2. Hence, for each $t_q \in \{t_{p+1}, t_{p+2}, \cdots, t_{k+1}\}$, constraints must be introduced to ensure that the cumulative remaining execution requirement of all jobs of criticality $\geq l$ with deadline at or prior to t_q can complete execution by t_q on a speed-s_l processor.

3. This is ensured by writing a constraint

$$\Big(\sum_{i | (\chi_i \geq l) \wedge (d_i \leq t_q)} \big(\sum_{j=p}^{q-1} x_{i,j} \big) \Big) \leq s_l (t_q - t_p). \tag{4.3}$$

   Note that for any job J_i with $d_i \leq t_q$, $\big(\sum_{j=p}^{q-1} x_{i,j}\big)$ represents the remaining execution requirement of job J_i at time instant t_p. The outer summation on the left-hand side is simply summing this remaining execution requirement over all the jobs of criticality $\geq l$ that have deadlines at or prior to t_q.

4. A moment's thought should convince the reader that rather than considering all t_q's in $\{t_{p+1}, t_{p+2}, \cdots, t_{k+1}\}$ as stated in (2) above, it suffices to only consider those that are deadlines for some job of criticality $\geq l$.

5. The Constraints (4.3) above only prevent missing deadlines after t_p when the (degraded) processor is continually busy over the interval between t_p and the missed deadline; what about deadline misses when the processor is not continually busy over this interval (and the right-hand side of the inequality of Constraints (4.3) therefore does not reflect the actual amount of execution received)? We point out that for such a deadline miss to occur, it must be the case that there is a subset of jobs of criticality $\geq l$ – those with release dates and deadlines between the last idle instant prior to the deadline miss and the deadline miss itself – that miss their deadlines on a speed-s_l processor. But this would contradict our assumption that the instance passes the necessary conditions of Lemma 4.3, i.e., all the jobs of criticality

Given MC instance $(\{J_1, J_2, \ldots, J_n\}, s_1, ..., s_m)$, with job release dates and deadlines partitioning the timeline over $[\min_i\{a_i\}, \max_i\{d_i\})$ into the k intervals $I_1, I_2, \ldots, I_k$
Determine values for the x_{ij} variables, $i = 1, \ldots, n, j = 1, \ldots, k$ satisfying the following **constraints:**

- For each i, $1 \leq i \leq n$, $\left(\sum_{j|t_j \geq a_i \wedge d_i \geq t_{j+1}} x_{i,j}\right) \geq c_i$. (4.1)
- For each j, $1 \leq j \leq k$, $\left(\sum_{i=1}^{n} x_{i,j}\right) \leq s_1(t_{j+1} - t_j)$. (4.2)
- For each p, $1 \leq p \leq k$, for each l, $2 \leq l \leq m$, and for each q, $p < q \leq (k+1)$

$$\Big(\sum_{i|(\chi_i \geq l) \wedge (d_i \leq t_q)} \big(\sum_{j=p}^{q-1} x_{i,j} \big) \Big) \leq s_l(t_q - t_p). \quad (4.3)$$

Figure 4.1: Linear program for constructing the scheduling table.

$\geq l$ together (and therefore, every subset of these jobs) execute successfully on a speed-s_l processor.

The entire linear program is listed in Figure 4.1, and the following steps of our linear programming based table-driven mixed-criticality scheduling approach TDMC-LP is described in Figure 4.2. Before proving its correctness and optimality, we first illustrate the proposed algorithm TDMC-LP by means of a simple example.

We can see that the run-time phase of TDMC-LP is performing a typical interval by interval execution – during run-time, unless an idleness is detected, no amount assigned in later intervals can be "promoted" – executed in the current interval. Moreover, according to Step 2a, we do not dismiss any amount when a processor degradation is detected.

Note that it is possible that after execution of some interval, some assigned amounts are not yet finished due to degradation. In such case, we *cannot* simply drop these amounts, but have to pass it over into the next interval. The reason for such additional modification during runtime is that Constraints (4.3) only provide guarantees to the total amount of required execution for each job *until its deadline*. This can be done by adding the unfinished part of the amounts into the corresponding

Given $J = \cup_{i=1}^{n} \{J_i\}$ to be scheduled on a varying-speed processor with speed thresholds $s_1, ..., s_m$:

- Construct the scheduling table S according to Figure 4.1, with $x_{i,j}$ denoting the assigned amount of execution to job J_i during the interval I_j, for each pair (i, j).
- **For** each interval I_j, $j = 1$ up to k:
  1. Higher-criticality execution is performed before lower-criticality ones within each interval, while amounts with the same criticality level may be executed in any order (e.g., in EDF order).
  2. At the end of the interval; i.e., $t = t_j$:
     (a) **If** t_j is some unfinished job's deadline, **then** the job is dropped; i.e., $\forall i$ that $d_i = t_j$, $x_{i,j} = -1$.
     (b) Other unfinished amounts (if exist) need to be passed over into the next interval; i.e., $\forall i$ that $d_i > t_j$, $x_{i,j+1} = x_{i,j+1} + x_{i,j} - Ex(i, j)$, where $Ex(i, j)$ denotes the executed amounts of job J_i within Interval I_j.
  3. Whenever an idleness is detected, we may execute the (released) jobs with amounts assigned to later interval(s) in the same priority order described in Step 1. 1.

Figure 4.2: Basic steps of the proposed scheduling algorithm TDMC-LP.

rows in the column of the scheduling table at the end of each interval (as described in Step 2b[2]. The necessity of such maintenance during run-time will also be shown in Example 4.4.

The execution order when an idleness is detected described in Step 3 has nothing to do with correctness — the proof of Theorem 1 will go through even the processor is left idled until the end of such interval.

Example 4.4. *We consider an MC instance I consisting of three jobs with parameters as depicted in Figure 4.3, with c_3's value left unspecified for now, and d_3 assumed to be larger than 5. The release dates and deadlines of these three jobs define three intervals: $I_1 = [0, 3)$; $I_2 = [3, 5)$; $I_3 = [5, d_3)$, as illustrated in Figure 4.3.*

Since there are three jobs in I ($n = 3$), Constraints (4.1) of the LP will be instantiated to the following three inequalities, specifying that all three jobs receive adequate execution in the

[2]Here $Ex(i, j)$ is *not* the total execution time of job J_i within Interval I_j — the processing speed during run-time needs to be considered (divided).

J_i	a_i	c_i	d_i	χ_i
J_1	0	3	5	1
J_2	2	1	5	2
J_3	0	c_3	d_3	3

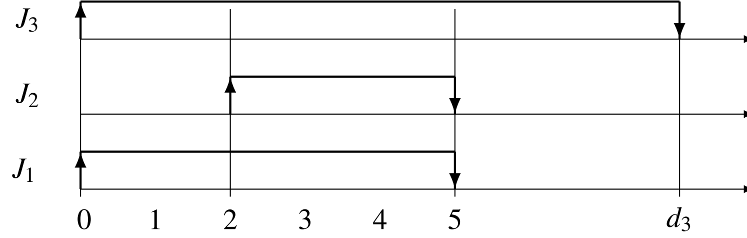


Figure 4.3: The MC job set considered in Example 4.4, with the graphical depictions.

scheduling table $S(I)$ *to execute correctly on a normal (non-degraded) processor:*

$$\begin{aligned} x_{11} + x_{12} &\geq 3; \\ x_{22} &\geq 1; \\ x_{31} + x_{32} + x_{33} &\geq c_3. \end{aligned}$$

There are also three intervals I_1, I_2*, and* I_3*. Constraints 4.2 of the LP will therefore yield the following three inequalities, specifying that the capacity constraints of the intervals are met:*

$$\begin{aligned} x_{11} + x_{21} + x_{31} &\leq 2; \\ x_{12} + x_{22} + x_{32} &\leq 3; \\ x_{13} + x_{23} + x_{33} &\leq d_3 - 5. \end{aligned}$$

It remains to instantiate the Constraints (4.3), that were introduced to ensure correct behavior in the event of processor degradation. In this example there are three criticality levels, and thus a need to consider degradation cases of both speed-s_2 *and speed-*s_3*. These must be separately instantiated*

to model the possibility of the processor degrading at the start of each of the three intervals I_1, I_2 and I_3. We consider these separately:

- **Degradation at the start of I_1.** *In this case, Constraints (4.3) is instantiated three times: speed-s_2 for $t_m = 5$, and both speed-s_2 and speed-s_3 for $t_m = d_3$:*

$$\begin{aligned} x_{21} + x_{22} &\leq (5-0)\,s_2; \\ \big(x_{21} + x_{22} + x_{23}\big) + \big(x_{31} + x_{32} + x_{33}\big) &\leq (d_3 - 0)\,s_2; \\ x_{31} + x_{32} + x_{33} &\leq (d_3 - 0)\,s_3. \end{aligned}$$

- **Degradation at the start of I_2.** *This case is similar as the above one that Constraints (4.3) is instantiated once for $t_m = 5$ and twice for $t_m = d_3$:*

$$\begin{aligned} x_{22} &\leq (5-2)\,s_2; \\ \big(x_{22} + x_{23}\big) + \big(x_{32} + x_{33}\big) &\leq (d_3 - 2)\,s_2; \\ x_{32} + x_{33} &\leq (d_3 - 2)\,s_3. \end{aligned}$$

- **Degradation at the start of I_3.** *In this case, Constraints (4.3) is instantiated twice, for $t_m = d_3$ with speeds s_2 and s_3:*

$$\begin{aligned} x_{33} &\leq (d_3 - 5)\,s_2; \\ x_{33} &\leq (d_3 - 5)\,s_3. \end{aligned}$$

(Note that there are nine variables and fourteen constraints in this particular example.)

Continuing with this example, suppose that c_3 and d_3 are 3 and 11 respectively, with degraded speeds $s_2 = 1/2$ and $s_3 = 1/3$. A possible solution to the LP would assign the x_{ij} variables the following values:

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 2 \end{bmatrix}.$$

As a consequence, the scheduling table would be as depicted in Figure 4.4.

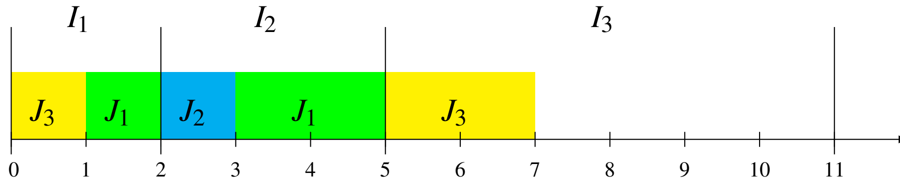


Figure 4.4: The constructed scheduling table of Example 4.4.

We can see that this scheduling table yields a correct scheduling strategy: observe that there are three contiguous blocks of execution of criticality-level 2 or greater: $[0,1)$, $[2,3)$, and $[5,7)$, and consider the possibility of the processor degrading during each:

- *If the processor degrades to speed-s_2 during $[0,2)$, then J_3 will execute over $[0,2)$ and $[5,9)$, while J_2 can execute over $[2,4)$. Both jobs of criticality ≥ 2 would thus meet their deadlines on the speed-$1/2$ processor. J_1 is executed over $[4,5)$ and dropped at $t = 5$.*

- *If the processor degrades to speed-s_3 during $[0,2)$, then for the first interval $[0,2)$, J_3 will be executed. However the assigned amount $x_{31} = 1$ may not be finished in the case processor degrades early, say at $t = 0$. As a result, the scheduling table needs to be updated at time $t = 2$ according to Step 2b in Figure 4.2: $x_{32} = 1/3$. Thus, J_3 will continuously execute over $[2,3)$ and $[5,11)$ and meet its deadline on the speed-$1/3$ processor. Time period $[3,5)$ will be used to execute J_2, and both J_1 and J_2 will be dropped at time $t = 5$ in the worst case, leaving x_{12} and x_{22} the value of -1 for reference. In the case processor degrades to speed-s_3 late, say at $t = 0.5$ (while remaining at unit-speed beforehand), the assigned amount $x_{31} = 1$ can be*

finished upon $t = 2$, and thus although under the slowest speed condition, J_2 may finish on time be executing over $[2,5)$.

- *If the processor degrades to a speed of either s_2 (or s_3) during* $[2,5)$*, then J_2 would execute prior to J_1 within this interval and gets finished on time. Job J_3 will not continue its execution until $t = 5$ since $x_{32} = 0$ — it only needs two additional units of execution which will be obtained by executing over the third interval* $[5,9)$.
- *If the processor degrades to speed-s_2 (or s_3) during* $[5,7)$*, J_3 will still meet its deadline since it has completed one unit of execution prior to the processor degradation — it needs two more units, which will be obtained by executing over* $[5,9)$ *(or* $[5,11)$*) on the speed-*$1/2$ *(or* $1/3$*) processor.*

We thus see that the solution of the LP does indeed yield a feasible scheduling strategy according to the proposed runtime strategies in TDMC-LP.

4.2.3 Properties of TDMC-LP

It is shown that Algorithm TDMC-LP is performing the "best-effort execution", and do not dismiss any amount when a processor degradation is detected. We now formally show that it is guaranteed the amounts with criticality level no lower than ℓ will be executed when processing speed remains no smaller than s_ℓ (which exactly maps to the correctness definition).

Theorem 4.5. *Algorithm TDMC-LP is correct.*

Proof: We prove by contradictory.

Assume some job J_i with criticality level χ_i is not finished by its deadline $d_i = t_q$ (at the end of Interval I_{q-1}), while the processor remains at (or above) a speed of s_{χ_i} over the period $[a_i, d_i)$.

From Constraint (4.3), we know that total assigned amounts with criticality level no lower than χ_i within Intervals $[a_i, d_i)$ can not exceed $s_{\chi_i}(d_i - a_i)$. Given the fact that no amount with lower criticality level(s) can be executed within period $[a_i, d_i)$ (or else J_i should be assigned and executed

during the period of such execution of lower criticality amounts), there must be some "carry-in" amounts with criticality level no lower than χ_i due to Step 2b.

Denote t_p as the end of the last interval (before a_i) with either idleness or some execution of amounts with criticality level lower than χ_i (so that no amount assigned before t_p with criticality level $\geq \chi_i$ can be "carried-in"). It is now evident that Constraint (4.3) must be violated for Interval $[t_p, t_q)$ under speed s_{χ_i}. □

Theorem 4.6. *Algorithm TDMC-LP is* optimal — *whenever it fails to maintain correctness, no other non-clairvoyant algorithm can.*

Proof: From Theorem 4.5, Algorithm TDMC-LP fails only when there is no feasible solution to the LP described in Figure 4.1. Since the three set of constraints are all necessary ones according to Lemma 4.3, violations of any of them indicates that the given instance is *not schedulable* under some circumstances (e.g., speed performances during run-time). Thus, no other algorithm can maintain correctness as well. □

Bounding the size of this LP. It is not difficult to show that the LP of Figure 4.1 is of size polynomial in the number of jobs n in MC instance I as well as the number of criticality levels m:

- The number of intervals k is at most $2n-1$. Hence the number of $x_{i,j}$ variables is $O(n^2)$.
- There are n constraints of the form (4.1), and k constraints of the form (4.2). The number of constraints of the form (4.22) can be bounded from above by (nkm), since for each $p \in \{1, \ldots, k\}$, there can be no more than n t_q's corresponding to deadlines of jobs. Since $k \leq (2n-1)$, it follows that the number of constraints is $O(n) + O(n) + O(n^2 m)$, which is $O(n^2 m)$.

Since it is known (Khachiyan, 1979; Karmakar, 1984) that a linear program can be solved in time polynomial in its representation, it follows that our algorithm for generating the scheduling tables for a given MC instance I takes time polynomial in the representation of I.

4.2.4 The Dual-Criticality Sub-Case

In this subsection, we consider a sub-case of the problem where only two criticality levels exist and derive a (computationally) more efficient algorithm (than the aforementioned linear program based one) for solving it. That is, we consider *dual*-criticality systems executing on a variable-speed processor characterized by just two speeds: a normal speed (assumed equal to 1) and a degraded speed (designated as s, with $s < 1$). We use the standard designations of LO and HI to denote the lower and higher criticality levels respectively.

4.2.4.1 A More Efficient Algorithm

At a high level, our algorithm is similar to LE-EDF described in Sec. 3.1.2: Given a dual-criticality MC instance I, we will first construct a *scheduling table* $S(I)$ (which can be viewed as a set of sub-jobs), and then make run-time job-dispatch decisions in a manner that is compliant with this scheduling table.

To construct the scheduling table, we first identify (Step 1 below) the latest time intervals during which the HI-criticality jobs must execute if they are to complete execution on a degraded processor; having identified these intervals, we construct (in Step 2) an EDF schedule for the HI-criticality jobs in these intervals.

Step 1. *Considering only the* HI*-criticality jobs in the instance, determine the intervals during which the jobs would execute upon a speed-s processor, if (i) each job executes for its* HI*-criticality WCET, and (ii) execution occurs as late as possible.*

It is evident that these intervals may be determined by considering the jobs in non-increasing order of their deadlines (i.e., latest deadline first), and taking the cumulative execution requirements of these jobs. These intervals may therefore be determined in $\Theta(n_{\text{HI}} \log n_{\text{HI}})$ time (which comes from the time complexity of EDF), where n_{HI} denotes the number of HI-criticality jobs.

Step 2. *Construct an EDF schedule for the* HI*-criticality jobs upon a preemptive processor that has speed s during the intervals determined in Step 1 above, and speed zero elsewhere.*

It follows from the optimality property[3] of EDF that if this step fails to ensure that each HI-criticality job receives adequate execution prior to its deadline, then no scheduling algorithm can guarantee correctness (see Definition 4.2) for this instance. We would therefore ***report failure***: this MC instance is not feasible. The remainder of this section assumes that Step 2 above was successful in completing each HI-criticality job prior to its deadline.

We now describe how to use this EDF schedule to construct the scheduling table — recall that this scheduling table is used for job dispatch decisions upon both the normal and degraded processor, and is therefore constructed assuming a normal-speed (i.e., speed-1) processor.

Step 3. *To construct the scheduling table, partition the timeline over* $[\min_i\{a_i\}, \max_i\{d_i\}]$ *into the* k *intervals* $I_1, I_2, \ldots, I_k$. *(Recall, from Sec. 3.2.2, that these are the intervals defined by the release dates and deadlines of all the jobs —* LO*-criticality and* HI*-criticality.)*

- *For each* HI*-criticality job* J_i *and each interval* I_ℓ *in which it is scheduled in the EDF schedule constructed in Step 2 above, execute* J_i *within this interval for an amount* x_{i_ℓ} *which equals to the amount of execution that* J_i *is allocated during Interval* I_ℓ *in the EDF scheduled constructed in Step 2 above.*

- *Assign* LO*-criticality jobs by simulating the EDF-scheduling of the* LO*-criticality jobs in the remaining capacity of the scheduling table — i.e., in the durations that are not already allocated to the* HI*-criticality jobs during Step 3.1 above.*

- *If during this EDF simulation there is any capacity left over within an interval (because the supply of currently-active* LO*-criticality jobs has been exhausted), then move over* HI*-criticality jobs, that had been assigned to the later intervals in the scheduling table during Step 3.1 above, into the current interval. In so doing favor earlier-deadline jobs over later-deadline ones.*

[3]Although the optimality proof of EDF in (Liu and Layland, 1973), which is based on a swapping argument, assumes that the processor speed remains constant, it is trivial to extend the proof to apply to processors that are only available during limited intervals, or indeed to arbitrary varying-speed processors.

J_i	a_i	c_i	d_i	χ_i
J_1	1	2	10	HI
J_2	5	1	8	HI
J_3	6	2	15	HI
J_4	0	4	6	LO
J_5	1	2	10	LO
J_6	10	3	13	LO

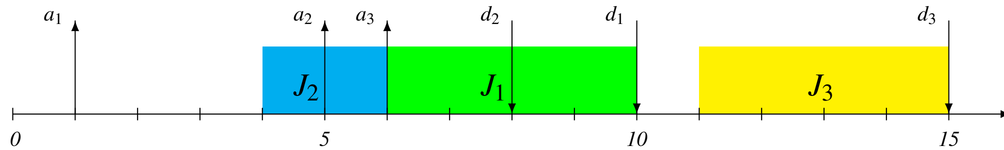
Figure 4.5: All MC jobs considered in Example 4.7, where a_i, c_i, and d_i stands for release date, WCET, and deadline respectively.

Note that Step 3.3 is not necessary for maintaining correctness. It is just one of the common choices when dealing idleness and/or pessimism in WCET estimations.

We illustrate this table construction process by means of the following example.

Example 4.7. *Consider the instance consisting of the six jobs J_1–J_6 shown in tabular form in Figure 4.5, to be implemented upon a processor of minimum degraded speed $s = 1/2$.*

In **Step 1***, we determine the intervals upon which the* HI*-criticality jobs J_1–J_3 would need to execute if they were to complete as late as possible, upon a degraded processor (one of speed-$1/2$); this is represented in the following diagram:*



In **Step 2***, we construct an EDF schedule of thel* HI*-criticality jobs J_1–J_3 upon a speed-$1/2$ processor. Lettting $x_{i,j}$ denote the amount of execution accorded to job J_i in interval I_j, the scheduling table $S(I)$ looks like this:*

I_j	$I_1 = [0,1)$	$I_2 = [1,5)$	$I_3 = [5,6)$	$I_4 = [6,8)$	$I_5 = [8,10)$	$I_6 = [10,13)$	$I_7 = [13,15)$
J_1	0	0.5	0	0.5	1	0	0
J_2	0	0	0.5	0.5	0	0	0
J_3	0	0	0	0	0	1	1

In **Step 3***, we now try to fill in this scheduling table with* LO*-criticality jobs, interval by interval.*

- *Interval* I_1 *will be filled with the job* J_4*.*
- *Both* J_4 *and* J_5 *are in Interval* I_2*;* J_4 *has the earlier deadline. As a result,* J_4 *receives* 3 *time units and* J_5 *takes the remaining* 0.5 *unit. Here we check that* J_4 *has received enough execution and meets its deadline.*
- *Interval* I_3 *has* 0.5 *units of execution remaining for job* J_5*.*
- *The remaining one time unit capacity in* I_4 *will be used by* J_5*. Until now the scheduling table for* HI*-criticality jobs has remained unchanged from the one constructed in Step 2 (and shown in the above table).*
- *For the Interval* I_5*, there is no active* LO*-criticality job, and the pre-allocated* HI*-criticality amount* $x_{1,5} = 1$ *can not fill this up. In this case, we try to move later-assigned* HI*-criticality amounts into this interval. Specifically, we consider the next interval* I_6*, where* x_{36} *should be "promoted" as* x_{35}*; i.e., the one time unit that originally belongs to Interval* $[10,13)$ *will be executed now. Note that after this step, the scheduling table for* HI*-criticality jobs is changed into the following one (with bold numbers highlighting changes).*
- *Interval* I_6 *is now empty and can be fully assigned to job* J_6*. Here we check that* J_6 *has received enough execution and meets its deadline.*
- *Nothing happens to Interval* $[13,15)$*.*

At the end of Step 3, the scheduling table for all jobs looks like this:

I_j	$[0,1)$	$[1,5)$	$[5,6)$	$[6,8)$	$[8,10)$	$[10,13)$	$[13,15)$
J_1	0	0.5	0	0.5	1	0	0
J_2	0	0	0.5	0.5	0	0	0
J_3	0	0	0	0	**1**	**0**	1
J_4	1	3	0	0	0	0	0
J_5	0	0.5	0.5	1	0	0	0
J_6	0	0	0	0	0	3	0

Computational complexity. Although an individual job in an EDF schedule for an instance of n jobs may be preempted as many as $(n-1)$ times, it is known (see, e.g., (Buttazzo, 2005)) that the *total* number of preemptions in any EDF schedule for an n-job instance cannot exceed $(n-1)$. In each column of the scheduling table, there should be at least one non-zero element unless all released jobs are finished beforehand. Each more non-zero element denotes that either a job is preempted, or a job finishes its execution within the corresponding interval. Since the number of total finishing points is fixed as $n_{\text{HI}} + n_{\text{LO}}$, the total preemption number cannot exceed $(n_{\text{HI}} + n_{\text{LO}} - 1)$, and number of total intervals is no greater than $(2n_{\text{HI}} + 2n_{\text{LO}})$, we know that the total number of non-zero entries in the table of Step 3 cannot exceed $(4n_{\text{HI}} + 4n_{\text{LO}} - 1)$, where n_{HI} (n_{LO}, respectively) denotes the number of HI-criticality (LO-criticality, resp.) jobs in the instance.

We note that standard techniques (see, e.g., (Mok, 1988)) for implementing EDF are known, that allow an EDF schedule for n jobs to be constructed in $\Theta(n \log n)$ time. Consequently, we conclude that the EDF-schedule of Step 2 can be constructed in $\Theta(n_{\text{HI}} \log n_{\text{HI}})$ time, and the total scheduler overhead during run-time is also bounded from above by $\Theta(n \log n)$ where $n = n_{\text{HI}} + n_{\text{LO}}$ denotes the total number of jobs.

4.2.4.2 An Optimization Version

Given a collection J of MC jobs and a degraded processor speed s, we have described how to obtain a correct scheduling strategy for the MC instance (J, s). We now consider an *optimization*

version of this problem: given the collection of MC jobs J, what is the *smallest* $\mathbf{s}$ such that there is a correct scheduling strategy for the instance $(J, \mathbf{s})$?

Lemma 4.3 gives us a lower bound: s can be no smaller than the speed of the slowest processor upon which the HI-criticality jobs in J would be correctly scheduled by EDF. But is this lower bound tight? The following example illustrates that it is not:

Example 4.8. *Consider the following three MC jobs:*

J_i	a_i	c_i	d_i	χ_i
J_1	0	2	2	LO
J_2	0	1	4	HI
J_3	2	1	4	HI

It is evident that (i) all three jobs are schedulable on a unit-speed processor (execute J_1 over $[0,2)$, J_2 over $[2,3)$, and J_3 over $[3,4)$), and (ii) J_2 and J_3 are schedulable on a speed-$\frac{1}{2}$ processor (execute J_2 over $[0,2)$, and J_3 over $[2,4)$). Hence, MC instance $(\{J_1,J_2,J_3\}, \frac{1}{2})$ satisfies the necessary conditions of Lemma 4.3. However, there is no (non-clairvoyant) scheduling strategy that can execute this instance correctly: consider the run-time behavior in which the processor operates in normal mode over $[0,2)$.

- *If J_1 did not execute exclusively over the interval $[0,2)$, then it misses its deadline at time instant 2. The processor remains in normal mode.*
- *If J_1 did execute exclusively over $[0,2)$, then the processor enters degraded mode at time instant 2.*

In either case, the instance was not correctly scheduled despite satisfying the necessary conditions of Lemma 4.3.

It turns out that a slight modification to the linear program of Figure 4.1 can be used to determine the smallest speed s: we simply add the objective function

$$\textit{minimize } s$$

to our linear program of Figure 4.1. That is, our modified linear program computes those values of the $x_{i,j}$ parameters that yield a scheduling strategy guaranteeing to meet all deadlines on a unit-speed processor, and HI-criticality jobs' deadlines when the degraded speed is *the smallest possible*; this smallest speed is the desired solution to the optimization version of our MC scheduling problem.

We have implemented this optimization algorithm and have conducted simulation experiments on randomly-generated MC instances to try and gain some insight into the trade-offs involved in MC scheduling upon varying-speed processors. We now describe these empirical investigations.

We generated a total of $30,000$ MC workload instances[4], for various different combinations of the four parameters described above. For each instance that we generated, we also computed two *load*[5] parameters — its HI-criticality load ($load_{\text{HI}}$) and its total load ($load_{\text{ALL}}$). Our observations are depicted in graphical form in Figures 4.6 and 4.7.

In Figure 4.6, the x-axes represent the HI-criticality load of the MC instance under consideration. The y-axis of the left graph represents the degraded speed s that the instance can tolerate, as computed by our optimization algorithm. By Lemma 4.3 the loading factor of the HI-criticality jobs is a lower bound on the degraded speed for which a correct scheduling strategy may exist — this lower bound is depicted as a dotted line in this graph, while the y-axis of the right graph represents the amount by which the computed degraded speed s exceeds this lower bound. Although we do not claim that our simulations are extensive or comprehensive enough to draw conclusions with absolute certainty, the evidence presented in these graphs does indicate that the actual minimum speed (as computed by our linear program) for which the typical randomly-generated MC instance is correctly schedulable, is very close to the lower bound implied by Lemma 4.3.

Figure 4.7 depicts the relationship between the *total* load of the instance, and the amount by which the computed degraded speed s exceeds the lower bound of Lemma 4.3. It is not surprising that s tends to diverge from the lower bound with increasing $load_{\text{ALL}}$: the intuition behind this is

[4]More details about our MC job generator can be found at Sec **??**.

[5]See, e.g. (Liu, 2000, p. 81) for the definition of the *load*, or *loading factor*, of a collection of jobs; it is known that the load is equal to the speed of the smallest processor upon which such a collection can be scheduled using preemptive EDF. For our instances, the HI-criticality load is the load of only the HI-criticality jobs in the instance, whereas the total load is the load of all the jobs in the instance.

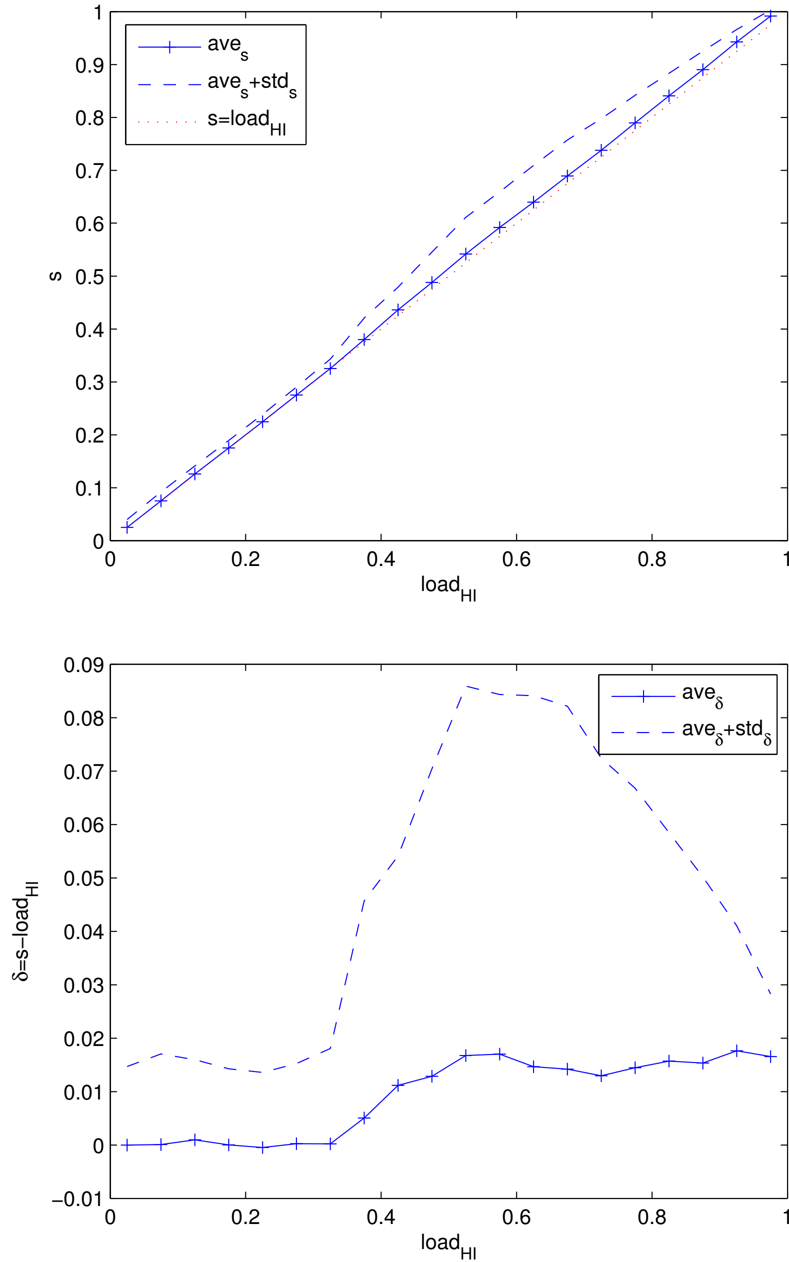


Figure 4.6: Degraded speed as a function of HI-criticality load and total load, where *ave* stands for average value and *std* stands for standard deviation

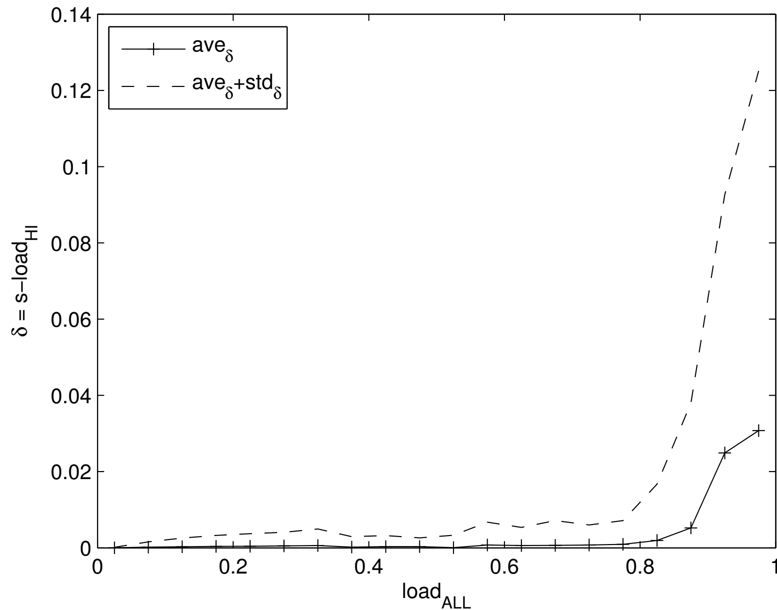


Figure 4.7: Degraded speed as a function of HI-criticality load, where *ave* stands for average value and *std* stands for standard deviation

that since the contribution of the LO-criticality jobs to $load_{\text{ALL}}$ also increases, LO-criticality jobs leave fewer time demands for the HI-criticality jobs to "extend" in degraded mode.

4.2.5 NP-Hardness for Non-Preemptive Scheduling

Recall that the TDMC-LP scheduling strategy we proposed in Sec. 4.2.2 works as follows. Given an instance I, we construct a scheduling table $S(I)$. During run-time scheduling decisions are initially made according to this table. If at any instant it is detected that the processor has transited to degraded mode, the scheduling strategy is *immediately* switched: henceforth, only HI-criticality jobs are executed, and these are executed according to EDF. Such a scheduling strategy requires that the job that is executing at the instant of transition can be preempted, and hence is not applicable for *non-preemptive* systems. In this subsection, we consider the problem of scheduling *non-preemptive* mixed-criticality instances.

Non-preemptive mandates that each job receives its execution during one contiguous interval of time. Let us suppose that a LO-criticality job is executing when the processor experiences a degradation in speed. We can specify two different kinds of non-preemptive requirements:

1. This LO-criticality job does not need to complete — it may immediately be dropped.

2. This LO-criticality job cannot be preempted and discarded — it must complete execution despite that fact that the processor has degraded and this job's completion is not required for correctness.

Although the first requirement – that the LO-criticality job may be dropped – may at first glance seem to be the more reasonable one, implementation considerations may favor the second requirement. For instance, it is possible that the LO-criticality job had been accessing some shared resource within a critical section, and preempting and discarding it would leave the shared resource in an unsafe state.

It has long been known (Lenstra et al., 1977) that the problem of scheduling a given collection of independent jobs on a single non-preemptive processor (that does not have a degraded mode) is already NP-hard in the strong sense (Lenstra et al., 1977)[6]. Since our mixed-criticality problem, under either interpretation of the non-preemptive requirements, is easily seen to be a generalization, it is also NP-hard. In fact, although determining whether an instance of (regular, not MC) jobs that all share a common release time can be non-preemptively scheduled on a fixed-speed processor is easily solved in polynomial time by EDF, it turns out that even this restricted problem is NP-hard for MC scheduling.

Theorem 4.9. *It is NP-hard to determine whether there is a correct scheduling strategy for scheduling non-preemptive mixed-criticality instances in which all jobs share a common release date.*

Proof Sketch: We prove this first for the second interpretation of non-preemptivity requirements (LO-criticality jobs that have begun execution must be executed to completion), and indicate how to modify the proof for the first interpretation.

[6]Indeed, it seems that it is difficult to even obtain *approximate* solutions to this problem, to our knowledge, the best polynomial-time algorithm known (Bansal et al., 2007) requires a processor speedup by a factor of 12.

This proof consists of a reduction of the partitioning problem (Garey and Johnson, 1979), which is known to be NP-complete, to the problem of determining whether a given non-preemptive mixed-criticality instance I can be scheduled correctly. The partitioning problem is defined as follows. *Given* a set S of n positive integers $y_1, y_2, \ldots, y_n$ summing to $2B$, *determine* whether there is a subset of S with elements summing to exactly B.

Given an instance S of the partitioning problem, we construct an instance of the mixed-criticality scheduling problem I composed of $(n+1)$ jobs $J_1, J_2, \ldots, J_{n+1}$. The parameters of the jobs are

$$J_i = \begin{cases} (0, y_i, 5B, \text{HI}), & 1 \le i \le n; \\ (0, B, 2B, \text{LO}), & i = n+1. \end{cases}$$

The normal processor speed is one; the degraded processor speed s is assigned a value equal to half: $s \leftarrow 1/2$.

We will show that there is a partitioning for instance S if and only if there is a correct scheduling strategy for I.

There is a partitioning for S. Let $S' \subseteq S$ denote the subset summing to exactly B. We construct our scheduling table as follows. Jobs corresponding to the elements in S' are scheduled over the interval $[0, B)$, after which J_{n+1} is scheduled over $[B, 2B)$, followed by the scheduling of the jobs corresponding to the elements in $(S \setminus S')$ over $[2B, 3B)$.

- If the processor enters degraded mode prior to time instant B, then only the HI-criticality jobs need to complete execution; it may be verified that they will do so by their common deadline.

- If the processor enters degraded mode over $[B, 2B)$, then J_{n+1} may execute for no more than the interval $[B, 3B)$. That still leaves adequate capacity for the jobs corresponding to elements in $(S \setminus S')$ to complete execution by their deadline at $5B$, on the speed-0.5 processor.

- Otherwise, J_{n+1} completes by time instant $2B$. That leaves adequate capacity for the jobs corresponding to elements in $(S \setminus S')$ to complete execution by their deadline at $5B$, regardless of whether the processor enters degraded mode or not.

There is *no* partitioning for S. In this case, consider the time instant t_o at which the LO-criticality job J_{n+1} begins execution. We consider three possibilities:

- If $t_o > B$, the processor remains in normal mode but J_{n+1} misses its deadline at $t = 2B$.
- If $t_o = B$, then the processor must have been idled for some time during $[0, B)$. If the processor were to now enter degraded mode at this time instant t_o, job J_{n+1} will execute over $[B, 3B)$, after which the strictly more than B units of remaining HI-criticality execution would execute — this cannot complete by the deadline of $5B$ on the speed-1/2 processor.
- Now suppose that $t_o < B$, and the processor enters degraded mode at this time instant t_o. It must be the case that $\leq t_o$ units of execution of the HI-criticality jobs has occurred prior to time instant t_o. Job J_{n+1} will execute over $[t_o, t_o + 2B)$, after which the at least $(2B - t_o)$ remaining units of HI-criticality work must complete. But on the speed-1/2 processor this would not happen prior to the time instant $t_o + 2B + 2(2B - t_o) = 6B - t_o > 5B$, which means that some HI-criticality job misses its deadline.

We have thus shown that there is a correct scheduling strategy for the non-preemptive mixed-criticality instance I if and only if S can be partitioned into two equal subsets.

The proof above assumed the second interpretation of non-preemptive requirements, in which LO-criticality jobs that begin execution need to complete even if the processor degrades. For the first interpretation of non-preemptive requirements (LO-criticality jobs do not need to complete if the processor degrades while they are executing), we would modify the proof by assigning the jobs $J_1, J_2, \ldots, J_n$ a deadline of $4B$ (rather than $5B$ as above). It may be verified that this modified MC instance can be scheduled correctly if and only if the S can be partitioned into two equal subsets. □

The intractability result of Theorem 4.9 implies that in contrast to the preemptive case, we are unlikely to obtain efficient (polynomial-time) optimal scheduling strategies for non-preemptive MC scheduling. In the future, we plan to work on devising, and evaluating, polynomial-time approximation algorithms for the non-preemptive and limited-preemptive scheduling of MC systems.

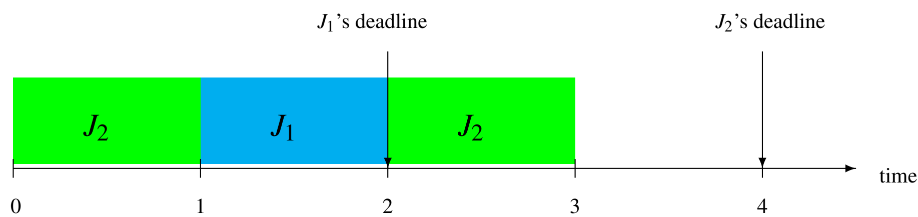
4.3 MC Job Scheduling on Non-Monitored Uniprocessor

We have considered in Sec. 4.2 the scheduling of MC job sets under the assumption that the platform upon which the workload is being executed is *self-monitoring* during run-time, in the sense that it immediately knows whether it transits from normal to degraded mode (i.e., if its speed falls from ≥ 1 to below s). In this section, we remove this assumption and consider platforms that lack the ability to self-monitor. We restrict our attention to the dual-criticality case in this section.

We first give the motivation in Sec. 4.3.1, then adapt the existing OCBP algorithm (see 3.1.3 for a detailed description) in Sec. 4.3.2, and finally quantifies the disadvantages of non-monitoring via speedup analysis Sec. 4.3.4. The work reported in this section shows one of the cases where existing algorithms can be adapted at no significant schedulability loss, as mentioned in our central thesis. Most of the contributions made in this section can be found at (Guo and Baruah, 2013).

4.3.1 Motivation

A natural question arises: does the lack of such self-monitoring ability even matter? We construct a simple example mixed-criticality instance below that shows that it does. This example instance consists of one LO-criticality job J_1 and one HI-criticality job J_2, that are to be preemptively scheduled on a processor with normal speed $s_n = 1$ and degraded speed $s_d = 0.5$. Both jobs arrive at time instant zero; J_1's WCET is 1 and its deadline is at time instant 2, while J_2's WCET is 2 and its deadline is at time instant 4. Upon a self-monitoring processor, we could start out scheduling the system according to the following scheduling table:



If at any instant during this execution the processor determines that its execution speed has degraded below 1, then J_1 is immediately discarded and the processor executes J_2 exclusively. It may be verified, by exhaustive consideration of all possible instants at which such degradation occurs, that this scheduling strategy will result in J_2 completing by its deadline regardless of when (if at all) the processor degrades, and in both deadlines being met if the processor remains normal (or degrades at any instant ≥ 2).

Suppose, however, that the processor *cannot* self-monitor: it does not know what its speed is at each instant during run-time. The schedule above is no longer correct: it is possible that the processor had degraded at the very beginning and was already operating at a reduced speed of 0.5 throughout the interval $[0,4)$, in which case neither job J_1 nor job J_2 would complete on time. This remains true even if J_2 were allocated for execution over $[3,4)$ upon it being discovered that it had not completed execution at time instant 3. Indeed, there will not be any scheduling strategy for this example instance that meets all our requirements (i.e., guarantees MC correctness) upon a non-monitoring processor, since the only scheduling strategy that can ensure that J_2 completes on a degraded processor would first execute J_2 to completion, but such a schedule would necessarily miss J_1's deadline even when the processor does not degrade.

Generally speaking, a self-monitoring processor knows it's degradation as soon as it occurs, and can make the best choice such as drop LO-criticality jobs to save enough capacity for HI-criticality jobs. However if the processor cannot self-monitor, it won't realize such degradation until a job received enough execution time and hasn't got finished — LO-criticality jobs will continue to get executed even when the processor is running at a degraded speed. Other than the monitoring of execution speed, the system model set up is exactly the same as described in Sec 4.2.1.

4.3.2 An OCBP Based Algorithm

We adapt the OCBP algorithm (Baruah et al., 2010b), which is described in Sec. 3.1.3, for scheduling such sets. As we take a different perspective of MC (which arises from execution speed),

here in this subsection we still describe our algorithm in full detail for the sake of completeness and clearness.

The high-level description of our algorithm is as follows. Given an MC instance $I = (J, s)$, we aim to derive offline (i.e., prior to run-time) a total priority ordering of the jobs of J such that scheduling the jobs according to this priority ordering constitutes a correct scheduling strategy, where *scheduling according to priority* means that at each moment in time the highest-priority available job is executed.

The priority list is constructed recursively using the approach commonly referred to in the scheduling literature as "Lawler's algorithm" (Lawler, 1973) or the "Audsley approach" (Audsley, 1991, 1993). We first determine (as described below) some job that may be assigned lowest priority, and assign it the lowest priority. Then the procedure is repeated to the set of jobs excluding the lowest priority job, until all jobs are ordered, or at some iteration a lowest priority job cannot be found.

Determining a lowest-priority job. It can be shown, using techniques very similar to those used in, e.g. (Baruah et al., 2010b), that if any LO-criticality job may be assigned lowest priority then so may the LO-criticality job with the latest deadline, and that if any HI-criticality job may be assigned lowest priority then so may the HI-criticality job with the latest deadline. Hence, we only need to determine whether one of the two jobs, the latest-deadline LO-criticality job or the latest-deadline HI-criticality job, may be assigned lowest priority.

- We assign the lowest priority to the latest deadline LO-criticality job if it would complete by its deadline on a speed-1 processor if every other job were assigned higher priority.
- Else, we assign the lowest priority to the latest deadline HI-criticality job if it would complete by its deadline *on a speed-s processor* if every other job were assigned higher priority.
- Else, we declare failure

J_i	a_i	c_i	d_i	χ_i
J_1	0	2	5	LO
J_2	0	3	10	HI
J_3	3	1	5	HI
J_4	2	4	10	LO

Figure 4.8: Mixed-criticality instance considered in Example 4.10.

(Note that at this point in time we do not check to determine whether the jobs assigned higher priority would meet their own deadlines or not — we are simply assuming that they each execute to completion in a work-conserving manner.)

We illustrate the priority assignment process by means of a simple example.

Example 4.10. *Consider the instance consisting of the four jobs J_1–J_4 shown in tabular form in Figure 4.8, to be implemented upon a processor of normal speed 1 and a degraded speed $s = 0.75$.*

- *It may be verified that J_4 would meet its deadline on a unit-speed processor if it were assigned lowest priority. We therefore assign J_4 the lowest priority.*
- *Next, we must determine which of the remaining three jobs may be assigned lowest priority amongst them.*
  - *If J_1 were assigned lower priority than both J_2 and J_3, then upon a unit-speed processor J_2 would execute over $[0,2)$, and J_2 and J_3 together would execute over $[2,4)$. That leaves J_1 just one unit of execution by its deadline, which is not enough to allow it to meet its deadline.*
  - *If J_2 were assigned lower priority than both J_1 and J_3, then on a speed-0.75 processor J_1 and J_3 would together execute for $(2+1)/0.75$ or 4 time units, over the interval $[0,4)$. That would allow J_2 to execute over the interval $[4,8)$ and consequently receive the required units of execution ($3/0.75 = 4$). We therefore assign J_2 the second-lowest priority from amongst the four jobs*

- *That leaves us with J_1 and J_3. Suppose J_1 is assigned lower priority than J_3. Then on a unit-speed processor J_1 would execute over $[0,2)$, and complete by its deadline. It may therefore be assigned the third-lowest priority.*

- *The remaining job J_3 is therefore assigned lowest priority.*

The final priority ordering is thus as follows (letting $J_i \rhd J_j$ denote that J_i has greater priority than J_j):

$$J_3 \rhd J_1 \rhd J_2 \rhd J_4$$

It is evident that this algorithm for assigning priorities is very efficient — it has a run-time that is a low-order polynomial in the number of jobs — and it is guaranteed to find a total priority ordering of the jobs, if one exists, such that scheduling according to this priority ordering is a correct online scheduling strategy.

Lemma 4.11. *Priority-based scheduling according to the priorities derived as described above constitutes a correct scheduling strategy.*

Proof: Suppose for a contradiction that our priority-assignment procedure was successful in assigning priorities to all the jobs in the instance $I = (J, s)$, but that job $J_i \in J$ misses its deadline during run-time.

- Suppose first that J_i is a LO-criticality job ($\chi_i = \text{LO}$). It follows from the manner in which priorities were assigned and the *sustainability* (Baruah and Burns, 2006) of preemptive fixed-priority scheduling with respect to processor speed, that J_i would have met its deadline despite the interference of jobs assigned greater priority, if the processor had executed throughout at a speed of 1 or greater. For the deadline miss to occur, hence, the processor must have executed at some speed strictly less than 1 at some instant prior to J_i's deadline. By the definition of correct scheduling strategy (Definition 4.2), J_i does not need to meet its deadline.

- Suppose now that J_i is a HI-criticality job ($\chi_i = \text{HI}$). It once again follows from the manner in which priorities are assigned, and the sustainability property, that J_i would have met

its deadline despite the interference of jobs assigned greater priority, if the processor had executed throughout at a speed of s or greater. For the deadline miss to occur, the processor must therefore have executed at some speed strictly less than s at some instant prior to J_i's deadline. By the definition of correct scheduling strategy (Definition 4.2), J_i does not need to meet its deadline.

We thus see that deadlines are missed only when doing so does not violate the requirements of correct scheduling. □

4.3.3 An Optimization Version of the Problem

Given an MC instance $I = (J, s)$, we derived above an algorithm for determining a correct scheduling strategy for instance I. An *optimization* version of the MC scheduling problem can also be defined:

- Given a collection of MC jobs J and a normal processor speed 1, what is the smallest degraded processor speed s such that we can determine a correct scheduling strategy for the MC instance $I = (J, s)$?

It is evident that both the optimization problem can be approximately solved to any desired degree of accuracy by applying the technique of "binary search" in conjunction with the algorithm for determining a correct scheduling strategy for a given instance (in which all parameters – J and s – are specified). Consider the optimization problem listed above, in which J is specified and the objective is to determine the smallest s. An upper bound on the value of s is 1; a lower bound is zero. We could therefore repeatedly guess a value for s within this interval, seeking the smallest value for which we are able to construct a correct scheduling strategy for $I = (J, s)$.

However, it turns out that we can in fact solve the problem directly, without needing to do a binary search. For the optimization problem listed above, the pseudo-code for doing so is given in Figure 4.9.

We start out "guessing" that the value of s is zero (line 2 of the pseudo-code), and repeatedly seeking to determine whether some job can be assigned lowest priority for this value of s. If so,

```
OPTI-1(J)
 1  J' ← J
 2  s ← 0
 3  repeat
 4    Let J_L be the latest deadline LO-criticality job in J'
 5    Let J_H be the latest deadline HI-criticality job in J'
 6    if J_L meets its deadline as the lowest-priority job on a speed-1 processor
 7    then  J_L gets the lowest priority
 8      J' ← J' \ {J_L}
 9    else
        Determine s', the smallest speed such that J_H meets its deadline as the
        lowest-priority job on a speed-s' processor
10      s ← max{s, s'}
11      J_H gets the lowest priority
12      J' ← J' \ {J_H}
13      s ← max{s, s'}
14  until J' is empty
15  return s
```

Figure 4.9: Determining the smallest degraded processor speed.

we continue; if not, we increase the guessed value of s to the smallest value needed to be able to assign some job the lowest priority and then continue. (If follows from the sustainability property of fixed-priority scheduling with respect to processor speed that if lower-priority jobs met their deadlines with the smaller values of s, they will continue to do so when s's value is increased.)

4.3.4 Quantifying the Benefits of Self-Monitoring

We now provide quantitative evaluations of the benefits of providing self-monitoring facilities to processors.

If an MC instance $I = (J, s)$ can be scheduled by a correct scheduling strategy upon a self-monitoring processor, then it is evident that the jobs in J can be scheduled by a correct scheduling strategy upon an unmonitored processor in which the normal and the degraded speeds are *both* equal to 1 (equivalently, the processor does not have a non-trivial degraded mode). The following lemma shows that this is the best general result we can come up with:

Lemma 4.12. *There are MC instances $I = (J, s)$ that can be scheduled by a correct scheduling strategy upon a self-monitoring processor, but for which (J, s') cannot be scheduled by a correct scheduling strategy upon an unmonitored processor for all $s' < 1$.*

In other words, such instances can only be scheduled upon an unmonitored processor if the processor does not have a non-trivial degraded mode.

Proof: We prove this lemma by demonstrating the existence of such an instance I. Let s be any constant less than one and k denote some large positive constant. Consider the collection of MC jobs listed in Table 4.1. For instance, if s were $1/2$ and k is chosen equal to 9, J_1 would have a WCET of 10 and a deadline at 20, while J_2 would have a WCET of 9 and a deadline at 18.

J_i	a_i	c_i	d_i	χ_i
J_1	0	$(k+1)$	$(k+1)/s$	HI
J_2	0	$k(1-s)/s$	k/s	LO

Table 4.1: An example of MC job set that is feasible upon a self-monitoring processor, while not schedulable upon an unmonitored one.

Upon a self-monitoring processor, we could construct a scheduling table that executes the HI-criticality job J_1 over $[0, k)$, J_2 over $[k, k/s)$, and J_1 again over $[k/s, k/s+1)$. For the example parameters of $s = 1/2$ and $k = 9$, this would correspond to scheduling J_1 over $[0, 9)$ and $[18, 20)$, and J_2 over $[9, 18)$.

It is evident that a self-monitoring processor would complete both jobs on a processor that executes throughout in normal mode. If the speed of the processor falls to below 1 at any instant, the LO-criticality job J_2 is immediately discarded and J_1 executed — it may be validated that this strategy results in J_1 always meeting its deadline as long as the processor speed remains at least s (for our example,$1/2$).

Upon a non-monitored processor with normal speed also equal to 1 and degraded speed s', we must execute J_2 for its WCET prior to its deadline (since we cannot determine, prior to J_2's deadline, whether the processor is in normal or degraded mode). Since J_1's deadline is after J_2's,

the duration for which J_1 will execute is hence bounded by

$$\begin{aligned} & (d_1 - a_1) - c_2 \\ & = \frac{k+1}{s} - \frac{k(1-s)}{s} \\ & = \frac{1+ks}{s} \end{aligned}$$

Suppose that the processor was to be degraded mode throughout; i.e., starting at time instant zero. For J_1 to execute to completion by its deadline, we need that

$$\begin{aligned} & s' \times \left(\frac{1+ks}{s}\right) \geq c_1 \\ \Leftrightarrow \quad & s' \times \left(\frac{1+ks}{s}\right) \geq (1+k) \\ \Leftrightarrow \quad & s' \geq \frac{s+ks}{1+ks} \end{aligned}$$

from which it follows that s' approaches one as $k \to \infty$. The lemma is thus proved. □

4.3.5 The Speedup Cost of Not Monitoring

As discussed in Sec. 2.4.3, much previous work on MC scheduling has focused on a model in which the processor speed is assumed to remain constant throughout run-time but each job is characterized by two different WCET values: a LO-criticality value and a larger HI-criticality value. An algorithm titled OCBP for *Own Criticality-Based Priorities* was proposed in (Baruah et al., 2010b, 2012a) for scheduling such MC systems, and the following *speedup bound* proved (as, e.g., (Baruah et al., 2010b, Lemma 5)): If an MC instance is schedulable on a given processor, then it is OCBP-schedulable on a processor that is $(1+\sqrt{5})/2$ (or approximately 1.618) times as fast.

Consider a single job with WCET of c upon unit speed processor. Upon an unreliable processor where $s(t)$ varies from $\bar{s}$ to 1, it may need up to c/s' units of time to finish execution. Under the assumptions of our model, a slower non-monitor processor can be transformed into longer WCET, and thus we can re-formulate the MC model that is described in Sec. 3.2.2 above into the Vestal

(2-WCET) model assumed by OCBP algorithm, in the following manner. Given an MC instance $I = (J, s)$ in the model described in Sec. 3.2.2, each job $J_i = (a_i, c_i, d_i, \chi_i)$ in J is modeled as a job J_i' with the same criticality, release date, and deadline, and with LO-criticality WCET equal to c_i and HI-criticality WCET equal to c_i/s. Hence for instance if $s = 1/2$, J_i''s LO-criticality WCET would equal c_i and its HI-criticality WCET would be equal to $2c_i$.

Upon such transformation, the algorithm that we described in Sec. 4.3.2 behaves in essentially the same manner as OCBP, and as a consequence similar speedup bounds can be derived: If an instance can be scheduled on a self-monitoring processor, then it can be scheduled on a non-monitoring processor that is $(1+\sqrt{5})/2$ times as fast in both the normal and the degraded mode.

Moreover, under the worst case that processor degraded to speed $\bar{s}$ and remains, all HI-criticality jobs will execute longer by a factor of $1/\bar{s}$. As a result, there becomes a fixed ratio between the 2 WCETs, and the following theorem shows a tighter bound comparing to previous work based on that.

Theorem 4.13. *Let $I = (J, s)$ denote an MC instance that can be correctly scheduled by an optimal scheduling strategy upon a self-monitoring processor. If the same job set J is not correctly scheduled by the algorithm described in Sec. 4.3.2 upon a platform with speedup ϕ, (i.e., a minimum* LO*-mode speed ϕ and degraded speed $\phi \times s$), then $\phi < \min\{2-s, \sqrt{s}+1\}$.*

Proof: For a given s, let $I = (J, s)$ denote some minimal instance that can be scheduled correctly by an optimal algorithm on a self-monitoring processor, but J is not correctly scheduled on a non-monitoring processor with LO-mode speed ϕ and degraded speed $\phi \times s$ using the algorithm of Sec. 4.3.2.

Let d^L denote the latest deadline of any LO-criticality job, and d^H the latest deadline of any HI-criticality job; let c^L and c^H denote the cumulative WCET's of the LO- and HI-criticality jobs respectively:

$$d^L = \max_{j \mid \chi_j = \text{LO}} d_j,$$

$$d^H = \max_{j|\chi_j=\text{HI}} d_j,$$
$$c^L = \sum_{j|\chi_j=\text{LO}} c_j,$$
$$c^H = \sum_{j|\chi_j=\text{HI}} c_j.$$

Consider now any work-conserving schedule of J upon a speed-ϕ processor, when each job J_i requests exactly c_i units of execution[7]. Let $\Lambda_1, \Lambda_2, \ldots$ denote the intervals, of cumulative length λ, during which the processor is idle in this schedule.

Observation 4.14. *No* LO*-criticality job has a scheduling window that overlaps with Λ_ℓ, for any ℓ.*

Proof: Suppose that some LO-criticality job J_i were to overlap withΛ_ℓ for some ℓ. This means that all the jobs which arrive prior to Λ_ℓ complete by the beginning of Λ_ℓ. Hence, J_i would complete by its deadline upon a speed-ϕ processor, if it were assigned lowest priority. But this contradicts the assumption that J is a *minimal* set that is not correctly scheduled on a non-monitoring processor using the algorithm of Sec. 4.3.2. □

Since J is assumed to be schedulable on a self-monitoring processor, all LO-criticality jobs would complete by d^L, the latest deadline of any LO-criticality job, on a speed-1 processor. It therefore follows from Observation 4.14 that the cumulative WCET's of all LO-criticality jobs cannot exceed $(d^L - \lambda)$:

$$c^L \leq d^L - \lambda \tag{4.4}$$

Since we are assuming that the instance J is not correctly scheduled by the algorithm described in Sec. 4.3.2 upon a platform with speedup ϕ, it must be the case that the LO-criticality job with the latest deadline cannot be the lowest-priority job on a speed-ϕ processor. Hence, it is necessary that

$$c^L + c^H > (d^L - \lambda)\,\phi \tag{4.5}$$

[7]We are not attempting to meet deadlines in this schedule, simply keeping the processor active whenever there are jobs remaining that have arrived but not completed execution, regardless of whether their deadlines are met or not.

We now argue from the schedulability of $I = (J, s)$ on a self-monitoring processor that

- All the jobs would complete by d^H, the latest deadline of any job, upon a speed-1 processor. Inequality 4.6 below, immediately follows.

$$c^L + c^H \quad \leq \quad d^H \tag{4.6}$$

- All HI-criticality jobs would complete by d^H upon a speed-s processor. Inequality 4.7 follows:

$$\frac{c^H}{s} \quad \leq \quad d^H \tag{4.7}$$

Observation 4.15. *Consider now any work-conserving schedule of J upon a speed-ϕ processor, when each* LO*-criticality job J_i executes for exactly c_i time-units, and each* HI*-criticality job J_i executes for exactly (c_i/s) time-units*[8]*. There are no idle intervals in this schedule.*

Proof: If there were an idle interval, any job whose scheduling window spans the idle interval would meet its deadline upon the speed-ϕ processor if it were assigned lowest priority. But this contradicts the assumption that J is a minimal instance that is not correctly scheduled on a non-monitoring processor with speedup ϕ using the algorithm of Sec. 4.3.2. □

Since we are assuming that J is not correctly scheduled on a non-monitoring processor with speedup ϕ using the algorithm of Sec. 4.3.2, it must be the case that the latest-deadline HI-criticality job will not meet its deadline if it were assigned the lowest-priority. Given Observation 4.15 above, it must then be the case that

$$c^L + \frac{c^H}{s} > d^H \ \phi \tag{4.8}$$

Suppose that the value of s is known, by multiplying both sides of Inequality (4.4) by a factor ϕ and combining with Inequality (4.5), we have

$$c^L + c^H > c^L \phi. \tag{4.9}$$

[8]As in Observation 4.14, we are not attempting to meet deadlines in this schedule.

By chaining Inequalities (4.8) and (4.6), we get

$$c^L + \frac{c^H}{s} > (c^L + c^H)\,\phi \tag{4.10}$$

while by chaining Inequalities (4.8) and (4.7), we get

$$c^L + \frac{c^H}{s} > \frac{c^H}{s}\,\phi \tag{4.11}$$

Let y denote the ratio of cumulative WCET length of different criticality jobs; i.e., $y := c^H/c^L$. From Inequalities (4.9)-(4.11), we conclude that

$$\phi < 1 + \min\{y, (1-s)/(y+s), s/y\}. \tag{4.12}$$

It is evident that $(1-s)/(y+s)$ and s/y decreases, with increasing $y \in \mathbb{R}^+$ and any fixed $s \in (0,1)$. Let $(1-s)/(y+s) = s/y$, and we have $y = s^2/(1-2s)$ which helps break Inequality (4.12) above into the following two inequalities:

$$\phi < 1 + \min\{y, (1-s)/(y+s)\}, \qquad \text{if } x < \frac{s^2}{1-2s} \tag{4.13}$$

$$\phi < 1 + \min\{y, \phi/s\}, \qquad \text{otherwise} \tag{4.14}$$

By solving the two equations $y = (1-s)/(y+s)$ and $y = s/y$, noticing that $y > 0$, we get solutions $y_1^* = (1-s)$ and $y_2^* = \sqrt{s}$. As a result, by substituting the min functions over y in Inequalities (4.13) and (4.14) and combining them together, we obtain the following relationship between ϕ and s:

$$\phi < 1 + \min\{1-s, \sqrt{s}\} = \min\{2-s, \sqrt{s}+1\}. \tag{4.15}$$

□

Figure 4.10 shows the bound on the speedup factor ϕ as a function of the degraded speed s (assuming normal speed of 1).

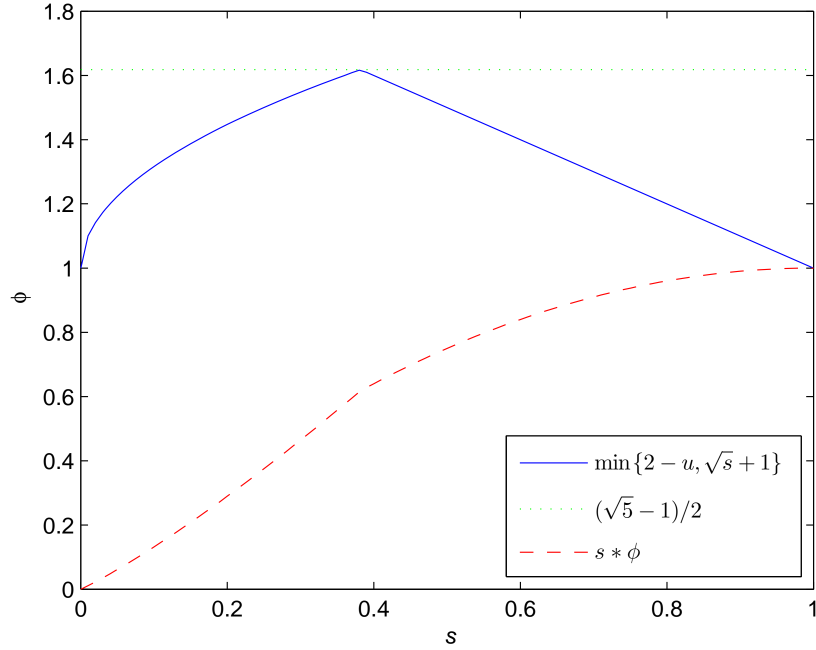


Figure 4.10: Lower bound on the speedup factor ϕ as a function of s - the degraded processor speed.

By solving the equation $2-s=\sqrt{s}+1$, we get $s^{*}=(3-\sqrt{5})/2$ and $\phi \leftarrow (2-s)=(1+\sqrt{5})/2$. Figure 4.10 shows the $(\sqrt{5}+1)/2$ speed-up factor upper bound, which matches the results in previous works.

Corollary 4.16. *Let $I=(J,s)$ denote an MC instance that can be correctly scheduled by an optimal scheduling strategy upon a self-monitoring processor. If the set J is not correctly scheduled by the algorithm described in Sec. 4.3.2 upon a platform with speedup ϕ, then $\phi<(1+\sqrt{5})/2$.*

From Figure 4.10, we can see that comparing to existing results, we can achieve a lower speed-up factor when the given s varies according to Theorem 4.13 for the non-self-monitoring case. The red slashed line shows how $s\times\phi$ is related to s. As the benefit of bringing in the algorithm, we would like to have it as low as possible - consider the case when $s\times\phi=1$, it means that for a degraded speed s, we need exactly a processor that runs $1/s$ times faster.

How *tight* is this relationship between speedup and s? To answer this question, consider the MC instance $I = (J, s)$; let σ denote $1/(1-s)$, and let J consist of the two jobs described in Table 4.2.

J_i	a_i	c_i	d_i	χ_i
J_1	0	1	1	LO
J_2	0	σ	$(\sigma - 1)$	HI

Table 4.2: An MC job set that demonstrates the tightness of the speedup bound ϕ shown in (4.15).

It has been shown (Baruah et al., 2012a, Proposition 2) that by taking $\sigma = (1+\sqrt{5})/2$, this instance reaches its schedulability bound. Noticing that for such σ, $s = (\sigma - 1)/\sigma = (3-\sqrt{5})/2$ takes exactly the value of s^* that is calculated above. This implies that Inequality (4.15) provides a tight bound for the speedup factor ϕ, and the upper bound of ϕ can be calculated by $\max(\phi) = 2 - s^* = 1 + \sqrt{s^*} = (1+\sqrt{5})/2$. We can also tell from Figure 4.10 that for a given $\phi \in (0,1)$, the upper bound of speedup factor varies from 1 when the ratio of normal to degraded speed is either zero or one, to $(1+\sqrt{5})/2$ when this ratio is equal to $(3-\sqrt{5})/2$ (or ≈ 0.382).

Note. Under the case that uncertainty arises solely from varying-speed platforms, the key difference between the non-monitored varying-speed processor model and the Vestal one is that all jobs will have the same ratio s between its optimistic execution bound (c^L) and the pessimistic one ($c^H = c^L/s$). This is the key reason why a tighter speedup result can be shown in this section. However, unlike the LP-based scheduler for the self-monitored case show in the previous section, here we are unable to achieve an optimal (i.e., speedup-1) scheduler for the non-monitored case — what we did in this section is adapted an existing algorithm (for scheduling Vestal job sets) and analyzed a tighter speedup bound for it under our model.

4.4 MC Job Scheduling on Multiprocessor

Embedded systems, especially safety-critical ones are increasingly implemented on multi-core platforms. Furthermore, as these multiprocessor platforms become more complex and sophisticated, their behaviors become less predictable. Larger variations will cause an increase of the pessimism to any conservative WCET-analysis tools.

Generally speaking, uniprocessor MC scheduling algorithms perform poorly on multiprocessor platforms. In this section, we seek to extend the LP based uniprocessor scheduler in Sec. 4.2 into (uniform) multiprocessor platform, while maintaining its optimality property. To the best of our knowledge, this is the only existing optimal MC multiprocessor scheduler. Most of the contributions made in this section can be found at (Guo and Baruah, 2014b).

4.4.1 Model and Preliminary Results

In this section, we study the scheduling of real-time *jobs* on m identical varying-speed processors that are characterized by a normal speed (without loss of generality, assumed to be 1) and a specified *degraded processor speed threshold* $s < 1$, under the following assumptions:

- Job preemption is permitted, with zero cost.
- Job migration is permitted, also with no penalty associated.
- Job parallelism is forbidden; i.e., each job may execute on at most one processor at any given instant.

Each *MC job* J_i is still characterized by a 4-tuple of parameters: a release date a_i, a WCET c_i, a deadline d_i, and a criticality level $\chi_i \in \{\text{LO}, \text{HI}\}$.

Let $s_i(t)$ denote the processing speed of processor i at time t, $i = 1, \ldots, m$. The interpretation is that the jobs in J are to execute on a multiprocessor system that has two modes: a *normal* mode and a *degraded* mode.

Although we have defined degraded mode (or HI-criticality mode) for varying-speed uniprocessor platform, it does not directly apply to multiprocessor platforms. Here we list two most reasonable definitions of degraded mode for multiprocessor platform:

Definition 4.17 (degraded mode)**.** *A system with m processors is in* degraded mode *at a given instant t if there exists* at least one *processor executing at a speed less than one; i.e.,* $\exists i, s_i(t) < 1$*; and moreover, all processors execute at a minimum speed of s; i.e.,* $\forall i, s_i(t) \geq s$*.*

Definition 4.18 (weak degraded mode)**.** *A system with m processors is said to be in* weak degraded mode *at a given instant if the processing speeds* $\{s_i\}$ *of all processors satisfy:*

$$\sum_{j=1}^{m} s_i \geq s \cdot m. \tag{4.16}$$

Under normal mode, *m* processors execute at unit-speed and hence each completes one unit of execution per unit time, whereas in degraded mode, according to the definition, each processor completes at least *s* units of execution per unit time. The weak degraded mode is called weak as it only requires the *m* processors are executing with an average speed no slower than *s*. It is not *a priori* known when, if at all, any of the processors will degrade: this information only becomes revealed during run-time when some processors actually begin executing at a slower speed.

Definition 4.19 (correct scheduling strategy)**.** *A scheduling strategy for MC instances is* correct *if it possesses the properties that upon scheduling any MC instance* $\mathscr{I} = (J, m, s)$*,*

- *if the system remains in normal mode throughout the interval* $[\min_i\{a_i\}, \max_i\{d_i\})$*, then all jobs complete by their deadlines;* **and**

- *if the system remains in normal mode or (weakly) degraded mode, then* HI*-criticality jobs (*J_i *with* $\chi_i = \text{HI}$*) complete by their deadlines.*

That is, a correct scheduling strategy ensures that HI-criticality jobs execute correctly regardless of whether the system runs in normal or (weakly) degraded mode; LO-criticality jobs are required to execute correctly only if all processors execute throughout in normal mode.

In this section, we will consider both definitions of degraded mode (in Secs. 4.4.3 and 4.4.4 respectively), and seek to determine *optimal scheduling strategy*:

Definition 4.20 (optimal scheduling strategy)**.** *An optimal scheduling strategy for MC instances possesses the property that if it fails to maintain correctness for a given MC instance $\mathscr{I}$, then no non-clairvoyant algorithm can ensure correctness for the instance $\mathscr{I}$.*

4.4.2 Step 1 — A Linear Program

The first step of our algorithms under either definition of the degraded mode is the same, which is constructing a linear program to determine the amount of execution to be completed for each job within each interval. Such assignment will possess the property that each job J_i receives c_i units of execution over its scheduling window $[a_i, d_i)$. The solution of the LP will be further used to construct schedules (in the following two subsections) under either definition of the degraded mode. Note that the linear program construction is very similar to the one described in Sec. 4.2.2 (which is for uniprocessor case). For the sake of completeness and clearness, in this subsection, we present the LP for multiprocessor in full detail.

Without loss of generality, assume that the HI-criticality jobs in I are indexed $1, 2, \ldots, n_h$ and the LO-criticality jobs are indexed $n_{h+1}, \ldots, n$. Let $t_1, t_2, \ldots, t_{k+1}$ denote the at most $2n$ distinct values for the release date and deadline parameters of the n jobs, in increasing order ($t_j < t_{j+1}$ for all j). These release dates and deadlines partition the time-interval $\left[\min_i\{a_i\}, \max_i\{d_i\}\right)$ into k intervals, which will be denoted as $I_1, I_2, \ldots, I_k$, with I_j denoting the interval $[t_j, t_{j+1})$.

To construct our linear program we define $n \cdot k$ variables $x_{i,j}$, $1 \leq i \leq n; 1 \leq j \leq k$. Variable $x_{i,j}$ denotes the amount of execution we will assign to job J_i in the interval I_j, in the scheduling table that we are seeking to build.

First of all, since no job can be executed on more than one processor in parallel, the following two sets of inequalities need to be introduced for ensuring no capacity constraint is violated:

$$0 \le x_{i,j} \le s(t_{j+1} - t_j), \forall (i,j), 1 \le i \le n_h, 1 \le j \le k; \tag{4.17}$$

$$0 \le x_{i,j} \le t_{j+1} - t_j, \forall (i,j), n_h < i \le n, 1 \le j \le k. \tag{4.18}$$

The following n constraints specify that each job receives adequate execution when system remains in normal mode:

$$\Big(\sum_{j | t_j \ge a_i \, \wedge \, d_i \ge t_{j+1}} x_{i,j} \Big) \ge c_i, \forall i, 1 \le i \le n. \tag{4.19}$$

The following k inequalities specify the capacity constraints of each interval:

$$\big(\sum_{i=1}^{n} x_{i,j} \big) \le m(t_{j+1} - t_j), \forall j, 1 \le j \le k. \tag{4.20}$$

It should be evident that any scheduling table generated in this manner from $x_{i,j}$ values satisfying the above constraints will execute all jobs to completion upon a normal-mode (non-degraded) system. It now remains to add constraints for specifying the requirements that the HI-criticality jobs complete execution even in the event of the system degrading into the faulty mode. It is evident that we only need to specify constraints for the most pessimistic degradation case — weakly degradation, where all processors run in total at the speed $s \times m$ (which holds true for "normal" degradation trivially, see Definitions 4.17 and 4.18).

Considering the case when *weakly* degradation occurs at the beginning of each interval, capacity constraints of each interval need to be specified for all HI-criticality amounts:

$$\big(\sum_{i=1}^{n_h} x_{i,j} \big) \le s \cdot m(t_{j+1} - t_j), \forall j, 1 \le j \le k. \tag{4.21}$$

It is not hard to observe that the worst-case scenarios occur when the system transits to weakly degraded mode at the very *beginning* of an interval — that would leave the maximum load of HI-criticality execution remaining to be done on the degraded system. For each ℓ, $1 \le \ell \le k$,

suppose that the degradation of the system occurs at time instant t_ℓ; i.e., the start of the interval I_ℓ. Henceforth, only HI-criticality jobs need to be guaranteed meeting deadlines. Thus for each possible deadline $t_m \in \{t_{\ell+1}, t_{\ell+2}, \cdots, t_{k+1}\}$, constraints must be introduced to ensure that the cumulative remaining execution requirement of all HI-criticality jobs with deadlines at or prior to t_m can complete execution by t_m on a system with m processors each of minimum speed s. This is ensured by the following constraint:

$$\Big(\sum_{i:(\chi_i = \text{HI}) \wedge (d_i \le t_m)} \big(\sum_{j=\ell}^{m-1} x_{i,j} \big) \Big) \le s \cdot m(t_m - t_\ell). \tag{4.22}$$

To see why this represents the requirement stated above, note that for any job J_i with $d_i \le t_m$, $\big(\sum_{j=\ell}^{m-1} x_{i,j}\big)$ represents the remaining execution requirement of job J_i at time instant t_ℓ. The outer summation on the left-hand side of Equation (4.22) is simply summing this remaining execution requirement over all the HI-criticality jobs that have deadlines at or prior to t_m.

A moment's thought should convince the reader that rather than considering all t_m's in $\{t_{\ell+1}, t_{\ell+2}, \cdots, t_{k+1}\}$ as stated above, it suffices to only consider those that are deadlines for HI-criticality jobs.

The entire linear program is listed in Figure 4.11. It is trivial that violating any of the constraints will result in *incorrectness* of the scheduling. Thus, we conclude that these conditions are *necessary*. If it could be further shown that they are also *sufficient*, we may conclude the *optimality* of our algorithm.

Note. Unlike the uniprocessor case studied in previous work (Sec. 4.2), to make these conditions sufficient here, we need to mimic a *processor-sharing* scheduling strategy. Discussions on converting the solution of LP into a correct schedule with processor-sharing will be provided in later parts of this section, and *optimality* will be shown based on the assumption that we can partition each interval into small enough quanta so that processor speed does not change inside each quantum.

Bounding the size of this LP. It is not difficult to show that the LP with linear constraints (4.17) - (4.22) is of size polynomial in the number of jobs n:

Given MC instance $\mathscr{I} = (J, m, s)$, with job release dates and deadlines partitioning the timeline over $[\min_i\{a_i\}, \max_i\{d_i\})$ into the k intervals $I_1, I_2, \ldots, I_k$.
Determine values for the x_{ij} variables, $i = 1, \ldots, n$, $j = 1, \ldots, k$ satisfying the following **constraints:**

- For each pair (i, j), $1 \leq i \leq n_h, 1 \leq j \leq k$,

$$0 \leq x_{i,j} \leq s(t_{j+1} - t_j).$$

- For each pair (i, j), $1 \leq i \leq n, 1 \leq j \leq k$,

$$0 \leq x_{i,j} \leq t_{j+1} - t_j.$$

- For each i, $1 \leq i \leq n$,

$$\Big(\sum_{j | t_j \geq a_i \ \wedge \ d_i \geq t_{j+1}} x_{i,j}\Big) \geq c_i.$$

- For each j, $1 \leq j \leq k$,

$$\big(\sum_{i=1}^{n} x_{i,j}\big) \leq m(t_{j+1} - t_j);$$

$$\big(\sum_{i=1}^{n_h} x_{i,j}\big) \leq s \cdot m(t_{j+1} - t_j).$$

- For each pair (ℓ, m), $1 \leq \ell \leq k$, $\ell < m \leq (k+1)$

$$\Big(\sum_{i:(\chi_i = \text{HI}) \wedge (d_i \leq t_m)} \big(\sum_{j=\ell}^{m-1} x_{i,j}\big)\Big) \leq s \cdot m(t_m - t_\ell).$$

Figure 4.11: Linear program for determining the amounts to be finished for each job within each interval.

- The number of intervals k is at most $2n-1$. Hence the number of $x_{i,j}$ variables is $O(n^2)$.

- There are n constraints of the forms (4.17) or (4.18), n constraints of the form (4.19), and $2k$ constraints of the forms (4.20) and (4.21). The number of constraints of the form (4.22) can be bounded by $(k \cdot n_h)$, since for each $\ell \in \{1, \ldots, k\}$, there can be no more than n_h of t_m's corresponding to the deadlines of HI-criticality jobs. Since $n_h \leq n$ and $k \leq (2n-1)$, it follows that the number of constraints is $O(n)+O(n)+O(n)+O(n^2)$, which is $O(n^2)$.

Since it is known that a linear program can be solved in time polynomial of its representation (Khachiyan, 1979) (Karmakar, 1984), our algorithm for generating the scheduling tables for a given MC job set J takes time polynomial in the representation of $|J|$.

4.4.3 Optimal Run-Time Strategy for Degraded Mode

In this subsection, we make the stronger restrictions about degraded mode, where no processor is allowed to execute at a speed slower than s (see Def. 4.17). One may argue that this is a rather restrictive definition, since we do not allow the case that a few processors to being nonfunctional, even when others execute at full speed. Sec. 4.4.4 discusses the case based on the alternative, less restrictive, definition to degraded mode.

Given a solution to the linear program constructed in the previous subsection, we now need to derive a run-time scheduling strategy that assigns an amount of execution $x_{i,\ell}$ to processors during the interval I_ℓ, for each pair (i,ℓ). According to the design of the linear program, run-time scheduling is now an interval-by-interval business — arrangements need to be made according to the table (calculated by the LP). We will show in this subsection how to mimic a processor-sharing schedule to execute mixed-criticality amounts *within each interval* in this possibly heterogeneous system (some processors may degrade while others may not at certain instants in time).

Within a given interval I_ℓ, we denote $f_{i,\ell} = x_{i,\ell}/(t-t_\ell)$ as the allocated fraction for a given amount $x_{i,\ell}$. According to Inequalities (4.20) and (4.21), we can derive the following bounds of these fractions:

$$f_{i,\ell} \leq s, \forall 1 \leq i \leq n_h; \tag{4.23}$$

$$f_{i,\ell} \leq 1, \forall n_h < i \leq n. \tag{4.24}$$

Definition 4.21 (lag)**.** *For any interval I_ℓ and an assigned amount $x_{i,\ell}$, its lag at any instant $t \in [t_\ell, t_{\ell+1})$ (within the interval) is given by:*

$$lag(x_{i,\ell}, t) = t \cdot f_{i,\ell} - executed(J_i, t). \tag{4.25}$$

Equation 4.21 defines a measurement to the difference between an ideal schedule and the actual execution of a given job. Under such a definition, we know that at any instant, non-negative *lag* for a job indicates that the schedule is correct *so far* with respect to this job. We will provide a strategy that guarantees zero *lag* at the end of each interval for all jobs while the system remains normal, and only for HI-criticality ones otherwise.

It should not be surprising that with (sufficient) preemption and migration, we can mimic a *processor-sharing* scheduling strategy that deals with this problem *correctly*. To mimic a processor-sharing scheduling strategy, jobs are simultaneously assigned fractional amounts of execution according to the solution of the LP. This can be done by partitioning the timeline into quanta of length Δ, where Δ is an arbitrarily small positive number. For each quantum, each job is executed for a duration of $f_{i,\ell} \cdot \Delta$, where $f_{i,\ell}$ has been defined to be the fraction of the job within Interval I_ℓ. In this way, by the end (and also at the beginning) of each quantum, *lag* for any job is zero, which leads to the correctness of the scheduling (thus far).

Now we have further reduced the original scheduling problem into the following: given a quantum of length Δ, m ordered processing speeds $\{s_1 \geq s_2 \geq ... \geq s_m \geq s\}$, and assigned fractions of mixed-criticality amounts $\{f_1, f_2, ..., f_n\}$, how to construct a feasible schedule on this heterogeneous system [9]? We can use the following algorithm to schedule the amounts for each quantum (with

[9] An important assumption is that changes to the speed of any processor *only* occur at quantum boundaries. In some sense this assumption is impractical. However, we may assume any processor's execution speed will not change dramatically within a short period (with length Δ). In this way, one can always "predict" how slow the processor can be in the near future. This pessimistic prediction will give us a lower bound on the execution speed of the following short period, and can serve as the "current" processor speed in our model.

length Δ), which is, in a larger picture, mimicking a processor-sharing schedule over the whole interval I_ℓ.

Without loss of generality, we assume that all fractions are sorted into decreasing order, and job IDs change accordingly for each quantum; i.e., $s \geq f_1 \geq ... \geq f_{n_h}$, and $1 \geq f_{n_h+1} \geq ... \geq f_n$

Algorithm Wrap-Around-MC(Δ, **f**)

- At the beginning of each quantum (with length Δ), sort both processor speeds $s_1, ..., s_n$ and assigned fractions $f_1, ..., f_n$ in decreasing order.

- Use slower processors to execute HI-criticality jobs. Consider HI-criticality fractions one by one in increasing order (smallest fit first), where a processor will not be used until all slower processors have been fully utilized (wrap-around).

- If the system is in the normal node; i.e., $s_i \geq 1, \forall i$, continue the "wrap-around" process for LO-criticality jobs on remaining faster processors.

- During execution, execute jobs on each processor following the same (priority) order of assignments in previous steps.

The following example shows how Wrap-Around-MC algorithm works.

Example 4.22. *Consider five jobs* $J_1 = J_2 = \{0, 0.4, 1, \text{HI}\}, J_3 = \{0, 0.5, 1, \text{HI}\}, J_4 = \{0, 0.3, 1, \text{LO}\}$, $J_5 = \{0, 0.7, 1, \text{LO}\}$*, to be scheduled on a platform of three varying-speed processors with degraded speed* 0.5*. Since all jobs share the same scheduling window, there is only one interval* $Ii = [0, 1)$*, and the LP has the solution* $x_{11} = x_{21} = 0.4, x_{31} = 0.5, x_{41} = 0.3, x_{51} = 0.7$*. Figures 4.12 and 4.13 show how Wrap-Around-MC would schedule these jobs under normal and a possible degraded mode respectively. For easier representation and understanding, we assume* $\Delta = 1$ *in the example without loss of generality — a smaller* Δ *would result in repeating of a shrinking version of the same schedule.*

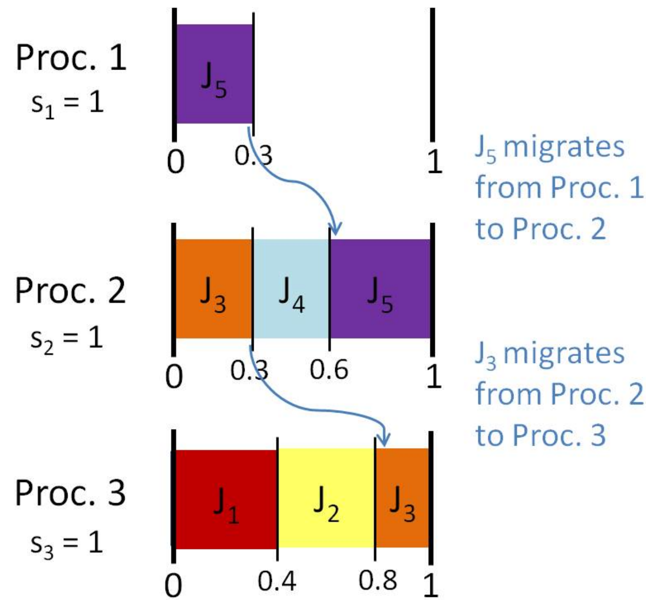


Figure 4.12: The schedule constructed by Wrap-Around-MC under normal mode in Example 4.22.

Theorem 4.23. *Algorithm Wrap-Around-MC (in addition to the linear program construction) is an* optimal *correct scheduling strategy for the preemptive multiprocessor scheduling of a collection of independent MC jobs.*

Proof: By *optimal*, we mean that if there exists a correct scheduling strategy (Definition 4.19 above) for an instance $\mathscr{I}$, then our scheduling strategy will succeed. From the definition, the obligation is to show that Wrap-Around-MC is able to correctly schedule any instance that can be correctly scheduled by any non-clairvoyant algorithm.

All inequalities defined in the linear program (4.17) – (4.3) have been shown to be necessary conditions. The optimality will come from the necessity of them — whenever Wrap-Around-MC returns fail, there must be some violations to the conditions, and thus no other non-clairvoyant algorithm can schedule this instance correctly.

What remains to be proved is this: given any solution to the LP, Algorithm Wrap-Around-MC will construct a correct scheduling strategy, so that these conditions are also sufficient.

We now show that parallel execution does not occur. In degraded mode, each processor remains a minimum execution speed of at least s. Since HI-criticality fractions are upper bounded by the

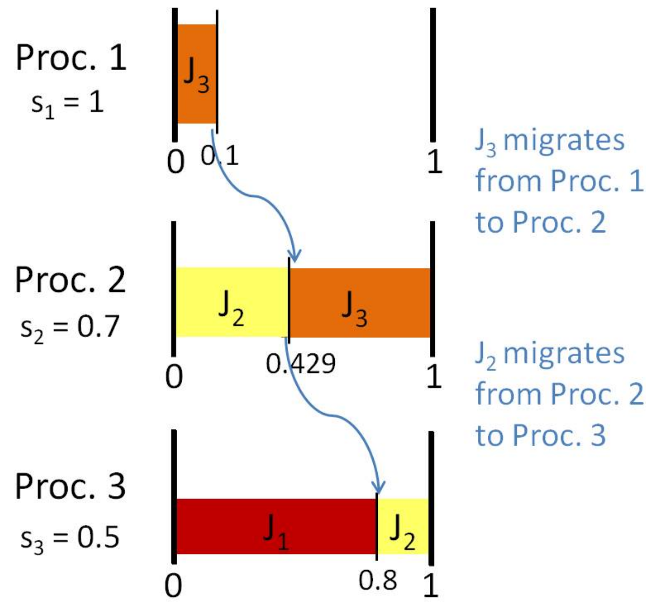


Figure 4.13: The schedule constructed by Wrap-Around-MC under a given degraded mode in Example 4.22.

same value (s), it is guaranteed that any HI-criticality job will not require a total execution time exceeding Δ. Thus with "wrap-around", the migrating jobs will not have any overlapping execution upon two different processors. A similar argument can be made regarding the LO-criticality jobs according to constraints (4.24) and normal mode processing speeds.

As far as each quantum follows Algorithm Wrap-Around-MC, the *lag* of all jobs remains zero under normal mode, while the *lag* of HI-criticality ones remains zero under degraded mode as well. From the definition of *lag*, we have shown that the conditions in LP are sufficient for the given algorithm to construct a *correct* schedule, and thus can conclude *optimality* of our algorithm.

□

The *optimality* of the algorithm tells us: (i) if all processors run in normal speed, all jobs will meet their deadlines; and (ii) if some (maybe all) processors run no slower than degraded speed s, HI-criticality jobs will meet their deadlines. We have not talked about how do deal with possible idleness during execution. Idleness is a critical issue in multiprocessor platforms, and is difficult to treat optimally in varying-speed systems. The following item may be added into Algorithm Wrap-Around-MC:

- Whenever some processor idles (which indicates this processor will remain idle for the rest of this quantum), execute the LO-criticality job with the earliest deadline that is assigned to next interval. If there is no LO-criticality job *active*, execute HI-criticality ones with similar attributes. Update the assigned value to further intervals by reducing the finished amount at the end of each interval.

Note that this item has nothing to do about the *optimality* of the algorithm; i.e., leaving any processor idle as it is according to Algorithm Wrap-Around-MC will still result in correctness.

4.4.4 Optimal Run-Time Strategy for Weakly Degraded Mode

So far we have focused on a rather restrictive model that places a relatively *strong* requirement on system behavior during degraded mode: all processors must execute at a minimum speed of s. The requirement is *strong* since we eliminate the case when only a few among m processors are not functional, while most ones execute at full speed — the whole system may still be able to ensure a cumulative speed of $s \cdot m$.

We now take the weaker definition of system degradation described in Def. 4.4.4. The requirement is considered *weak* because if the m processors are executing with an average speed no slower than s, correctness must be guaranteed for HI-criticality jobs. Now it includes the annoying case that several processors may run at a very low (but not zero) speed, and they need to be well utilized for some heavy load instances.

The following simple example shows how Algorithm Wrap-Around-MC will fail in weak degraded mode for a *feasible* job set.

Example 4.24. *Consider two jobs* $J_1 = J_2 = \{0, 0.5, 1, \text{HI}\}$*, to be scheduled on a varying-speed platform of two processors with degraded speed* 0.5*. Since both jobs share the same scheduling window, solution* $x_{11} = x_{12} = 0.5$ *to the LP is trivial.*

Now consider the case if at the very beginning Processor 1 degrades into speed 0.75*, while Processor 2 degrades into speed* 0.25*. Although the system is no longer in degraded mode; it*

still satisfies the weak degradation definition. Figure 4.14 compares the incorrect result by Wrap-Around-MC (where the dotted box marks the parallel execution period) and a possible correct scheduling strategy.

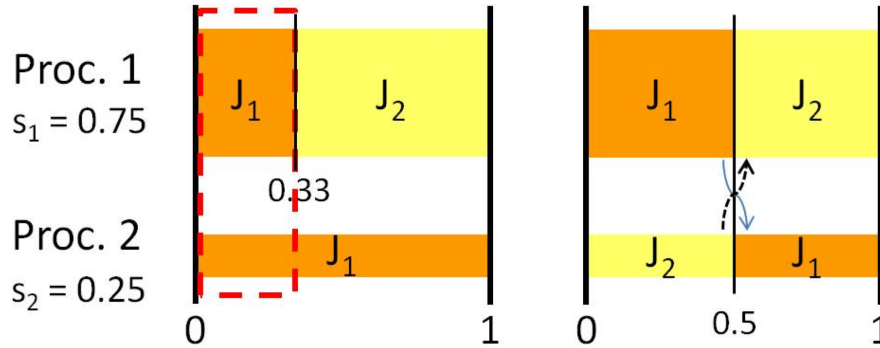


Figure 4.14: The incorrect schedule constructed by Wrap-Around-MC (left), and a feasible one (right) under weak degraded mode in Example 4.24.

This example shows that wrap-around is no longer optimal under weak degraded mode. Additional "slices" inside each quantum need to be made, so that jobs will migrate and get rid of parallel execution. In general, for optimal scheduling on this kind of heterogeneous system, studies have been made, and the current state of art suggests the adaptation of the Level Algorithm (Horvath et al., 1977).

The Level Algorithm creates a significantly large number ($O(m^2)$) of preemptions and migrations for each short period (quantum), in order to fully utilize all slow processors with jobs that need to execute for a considerable duration during this quantum without running into the parallel execution problem. With the help of (the optimal) Level Algorithm, we can extend Algorithm Wrap-Around-MC as follows to correctly deal with systems in weak degraded mode.

Algorithm Level-MC(Δ, **f**)

- At the beginning of each quantum (with length Δ), order both the processor speeds $s_1, \ldots, s_n$ and the assigned fractions $f_1, \ldots, f_n$ in decreasing order.
- **If** the system is in normal mode, "wrap-around" all jobs.
- **Elseif** the system is in degraded mode, "wrap-around" HI-criticality jobs.

- **Elseif** the system is in weak degraded mode, apply the Level Algorithm to HI-criticality jobs.
- During run-time, in both the normal and the degraded modes, jobs are assigned the priority order same as the assignment order in the steps above, and are executed on their allocated processors. In the weak degraded mode, priorities of jobs are not fixed, and the detailed schedule is given by the Level Algorithm.

The following example illustrates how Level-MC works under weak degraded mode.

Example 4.25. *Consider four* HI*-criticality jobs* $J_1 = \{0, 0.2, 1, \text{HI}\}, J_2 = \{0, 0.25, 1, \text{HI}\}, J_3 = \{0, 0.4, 1, \text{HI}\}, J_4 = \{0, 0.5, 1, \text{LO}\}$*, to be scheduled on a platform of three varying-speed processors with a minimum weak degraded speed threshold of* 0.5*. Consider the weak degraded case where three processors run at speeds of* $0.3, 0.4,$ *and* 0.8*, respectively (the average speed of the system is* 0.5*).*

Since all jobs share the same scheduling window, there is only one interval $I_i = [0, 2)$*, and the LP has the solution* $x_{11} = 0.2, x_{21} = 0.25, x_{31} = 0.4,$ *and* $x_{41} = 0.5$*. Figure 4.15 shows how Level-MC would schedule these jobs under such weak degraded mode for the next quantum. Without loss of generality, we assume unit length for each quantum; i.e.,* $\Delta = 1$*. A shorter quantum length would result in repeating of a shrunk version of the same schedule pattern.*

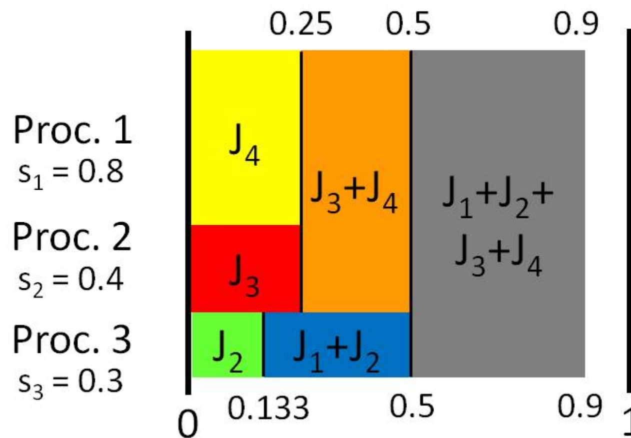


Figure 4.15: The schedule constructed by Level-MC under weak degraded mode in Example 4.25.

In the schedule shown in Figure 4.15, jobs are jointly *executing on more than one processor during some intervals; e.g., jobs J_1 and J_2 during interval* $[0.133, 0.5)$*, jobs J_3 and J_4 during interval* $[0.25, 0.5)$*, and all jobs during interval* $[0.5, 0.9)$*. The Level Algorithm designs the schedule in a way that capacity, as well as execution speeds, are evenly divided (shared) by combined jobs. The intuition is that since a heavy job executes on a high-speed processor, there may be an instant that two (or more) jobs have the same amount left (to be executed). For example, in the schedule given by Figure 4.15, both J_1 and J_2 require* 0.2 *time units of further execution at time $t = 0.133$. From then on, they should execute at the same speed, and thus are* joined *by the Level Algorithm.*

To jointly *execute n jobs on m processors, where $n \geq m$, the Level Algorithm divides the period into n equal sub-periods, and makes the assignment that each processor executes (and only executes) each job for one subperiod. Figure 4.16 expresses the schedule for all the jobs by this divide-and-assign scheme.*

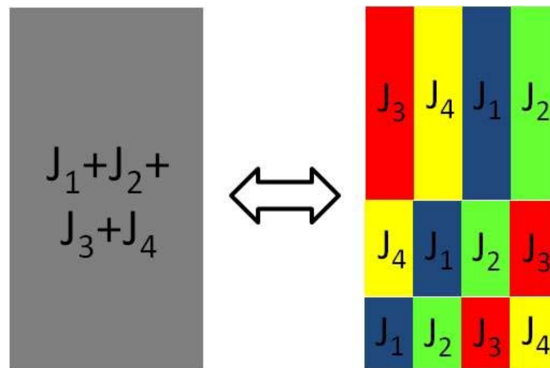


Figure 4.16: Joint execution of all jobs on the system by Level Algorithm during $[0.5, 0.9)$ of Example 4.25.

Theorem 4.26. *Algorithm Level-MC (in addition to the linear program construction) is an* optimal *correct scheduling strategy for the preemptive multiprocessor scheduling of collections of independent MC jobs.*

Proof: Similar to the proof of Theorem 4.23, we only need to show that the weak degraded condition is also *sufficient* for Level-MC to construct a correct schedule.

According to (Horvath et al., 1977), the Level Algorithm will always return a feasible schedule if the following m constraints hold (assume both $\{f_i\}$ and $\{s_j\}$ are in *decreasing* order):

$$\sum_{j=1}^{i} f_j \leq \sum_{j=1}^{i} s_j, \forall i, 1 \leq i \leq m-1; \tag{4.26}$$

$$\sum_{j=1}^{n_h} f_j \leq \sum_{j=1}^{m} s_j. \tag{4.27}$$

From Inequality (4.23), we have $f_j \leq s$, for any j. Since $\{s_j\}$ are ordered in decreasing order, from the property of "average", we know that $s \cdot i \leq \sum_{j=1}^{i} s_j$ holds true for any i. Putting these together, we have Inequality (4.26). Inequality (4.27) follows directly from the capacity constraint (4.21).

As a consequence, under such a processor-sharing protocol, the Level Algorithm returns a feasible schedule within each quantum (a small enough interval of length Δ). Here feasibility indicates that no job gets executed simultaneously on more than one processor, and all jobs receive their designated amounts by the end of the quantum. As the system continues to run quantum by quantum, the HI-criticality amounts are guaranteed to be finished by their assigned fractions (with zero *lag*). This indicates all HI-criticality jobs will meet their deadlines when the system is in weak degraded mode.

Correctness in both normal mode and degraded mode (each with a processing speed no less than s) follows from Theorem 4.23 since no change has been made from Algorithm Wrap-Around-MC for these cases.

We have shown that Inequalities (4.17) — (4.3) are *sufficient* for Algorithm Level-MC to construct a *correct* schedule. Since it has been shown that these conditions are also necessary, we can conclude the *optimality* of the algorithm. □

Dropping LO-criticality jobs. Under the definition of *correctness*, the two algorithms proposed so far drop all LO-criticality jobs whenever degradation occurs (even to only one of the processors). One can certainly argue that such sacrifice may not be necessary.

Inequalities (4.26) and (4.27) can also be applied to all jobs (instead of only HI-criticality ones) to check the feasibility of the current system (described by processing speeds). The following item can be added into Algorithms Wrap-Around-MC and Level-MC to further improve them by not dropping the LO-criticality jobs in some of the degraded cases:

- If the system is in (weak) degraded mode, check feasibility conditions for all jobs; i.e., Inequalities (4.26) and (4.27). If they hold, apply the Level Algorithm to all jobs; else follow the previous protocols for the HI-criticality jobs only, and suspend the LO-criticality ones.

However, whether *optimality* can be proved under such protocol remains unknown; i.e., if our algorithm drops any LO-criticality job under certain degradation condition(s), is it necessarily the case that other algorithm(s) must drop some LO-criticality job(s) to guarantee correctness?

4.4.5 Necessity of Processor-Sharing

We show that processor-sharing (a technique used in our algorithms described in above subsections) is *necessary* in order to ensure that any instance for which the LP generates a solution can be scheduled during run-time.

In the following example, the mixed-criticality instance is composed of three jobs and two processors. We will show that although the Linear Program has a feasible solution for this instance, there does not exist a feasible schedule for this job set without processor-sharing. Note that it suffices to show the case under stronger restrictions of degraded mode, as it can be viewed as a special case for the weakly degraded mode.

Example 4.27. *Consider three independent jobs* $J_1 = J_2 = \{0, 1, 2, \text{HI}\}, J_3 = \{0, 2, 2, \text{LO}\}$*, to be scheduled on a platform of two varying-speed processors with degraded speed* 0.5*. Since all jobs share the same scheduling window, there is only one interval* $I_i = [0, 2)$*, and the LP has the solution* $x_{11} = x_{21} = 1$ *and* $x_{31} = 2$*.*

Wrap-Around-MC will execute this set of jobs easily by combining the HI-criticality ones together and executing them on one processor while the system remains normal. Job J_3 *can be*

dropped whenever the system begins to suffer from degradation. Here processor-sharing gives us the ability to execute any fraction of a job within a short enough quantum (with length Δ).

Under the case where processor-sharing is forbidden, we can still assume processors do not *change their speeds during each quantum. The only difference is that we can no longer assign a fraction of capacity to each quantum; one certain job needs to be assigned to a given processor within each quantum. We will show that no matter how small Δ is, there does not exist a feasible schedule for this job set (without processor-sharing).*

Consider two possible decisions at time $t = 0$ (for the next quantum) — we may either assign both two processors the HI*-criticality jobs, or allocate the* LO*-criticality job to one of them.*

The first choice is certainly not correct in the case both processors never degrades. To make sure the LO*-criticality job with a utilization of 1 meets its deadline, it needs to be executing for the whole interval. However, the* LO*-criticality job will not start to execute until $t = \Delta$ under this decision. Since a job cannot be executed on both processors in parallel, remaining capacity $2-\Delta$ on either processor is not enough for the* LO*-criticality job to meet its deadline.*

For the second choice, consider the case that both processors degrade into 0.5-speed at instant $t = \Delta$. There remains a HI*-criticality job (assumed to be J_2, without loss of generality) which requires an execution of 1 time unit within the interval. However, each processor has a remaining capacity of $(2-\Delta)\cdot 0.5$, which is smaller than 1. Since a job cannot be executed on both processors in parallel, the remaining capacity on either processor is not enough for J_2 to be finished on time. The wasted Δ capacity (used for executing the* LO*-criticality job) of the system is crucial (and unavoidable).*

The problem without processor-sharing is that we can no longer guarantee that upon any instant that the system may degrade, we will execute a fraction of 0.5 to both HI-criticality jobs on one processor. The *lag* of some HI-criticality job may be negative, which means the constructed schedule is "left behind" when compared to the ideal case. The key assumption in processor-sharing is that processor speed will not change throughout each quantum. This gives us the ability to execute each job a proper length which leaves a zero *lag* after each period of length Δ.

4.5 MC Task Scheduling on Uniprocessor

In Secs. 4.2-4.4 above, we have considered mixed-criticality (MC) systems that can be modeled as finite collections of jobs. However, many real-time systems are better modeled as collections of *recurrent processes* that are specified using, e.g., the sporadic tasks model described in Sec. 2.1.3. In this section, we briefly consider this more difficult problem of scheduling mixed-criticality systems modeled as collections of sporadic tasks under the varying-speed interpretation of MC systems. As some initial efforts, we choose to target the *uniprocessor* case in this section, and left multiprocessor for future work. Most of the contributions made in this section can be found at (Baruah and Guo, 2013) and (Guo and Baruah, 2014a).

As with traditional (i.e., non MC) real-time systems, we model an MC real-time system τ as being composed of a finite specified collection of MC recurrent tasks, each of which will generate an unbounded number of MC jobs. We restrict our attention here to *dual-criticality* systems of *implicit-deadline MC sporadic tasks*, where each task is characterized by a 3-tuple of parameters: $\tau_i = (C_i, T_i, \chi_i)$. The quantity $U_i = C_i/T_i$ is referred to as the *utilization* of τ_i.

An *implicit-deadline MC sporadic task system* is specified by specifying a finite number $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of such sporadic tasks, and the degraded processor speed $s < 1$ (it is assumed that the normal processor speed is one without loss of generality). Such an MC sporadic task system can potentially generate unbounded number of different MC instances (collections of jobs), each instance being obtained by taking the union of one sequence of jobs generated by each sporadic task.

If *unbounded preemption* is permitted, then the scheduling problem for implicit-deadline MC sporadic task systems on uniprocessors is easily and efficiently solved in an optimal manner. We first derive (Theorem 4.28) a necessary condition for the existence of a correct scheduling strategy. We then present a scheduling strategy, *Algorithm preemptive-MC*, and prove (Theorem 4.29) that it is optimal.

Theorem 4.28. *A necessary condition for MC sporadic task system* (τ, s) *to be schedulable by a non-clairvoyant correct scheduling strategy is that*

1. *the sum of the utilizations of all the tasks in* τ *is no larger than* 1*, and*

2. *the sum of the utilizations of the* HI*-criticality tasks in* τ *is no larger than s.*

Proof: It is evident that the first condition is necessary in order that all jobs of all tasks in τ complete execution by their deadlines upon a normal processor, and that the second condition is necessary in order that all jobs of all the HI-criticality tasks in τ complete execution by their deadlines upon a degraded (speed-s) processor. □

In order to derive a correct scheduling strategy, we first observe that using preemption we can mimic a *processor-sharing* scheduling strategy, in which several jobs are simultaneously assigned fractional amounts of execution with the constraint that the sum of the fractional allocations should not exceed the capacity of the processor. This can be done by partitioning the timeline into intervals of length Δ where Δ is an arbitrarily small positive number, and using preemption within each such interval to ensure that each job that is assigned a fraction f of the processor capacity gets executed for a duration $f \times \Delta$ within this interval.)

Consider now the following processor-sharing scheduling strategy:

Algorithm Preemptive-MC.

1. Initially (i.e., on the normal –non-degradation– processor), assign a share U_i of the processor to each task τ_i during each instant that is active[10].

2. If the processor transits to degraded mode at any instant during run-time, immediately discard all LO-criticality tasks and execute the HI-criticality tasks according to EDF.

Theorem 4.29. *Algorithm preemptive-MC is an optimal correct scheduling strategy for the preemptive uniprocessor scheduling of MC sporadic task systems.*

[10] A task is defined to be *active* at a time instant t if it has released a job prior to t and this job has not yet completed execution by time t.

Proof: Let τ denote an MC implicit-deadline sporadic task system satisfying the necessary conditions for schedulability that have been identified in Theorem 4.28.

It is evident that Algorithm preemptive-MC meets all deadlines if the processor operates at its normal speed, since the processor-sharing schedule ensures that each job of each task τ_i receives exactly C_i units of execution between its release date and its deadline.

Suppose that the processor degrades at some time instant t_o. If we were to immediately discard all LO-criticality tasks, the second necessary schedulability condition of Theorem 4.28 ensures that there is sufficient computing capacity on the degraded processor to continue a processor-sharing schedule in which each HI-criticality task τ_i with an active job receives a share U_i of the processor. The correctness of Algorithm preemptive-MC now follows from the existence of this processor-sharing schedule, and the optimality property of preemptive uniprocessor EDF. □

If preemption is forbidden, then scheduling of MC sporadic task systems becomes a lot more challenging. As with the collections of independent jobs (Theorem 4.9), this problem, too, can be shown to be highly intractable.

4.6 Summary

In this chapter, we propose a new interpretation of MC scheduling, where MC arises (solely) from varying-speed platforms. Under this model, a single WCET threshold will be assigned to a single piece of code, yet its actual run-time is related to the performance of the platform, which remains unknown *a priori*. The mode switch of the system is triggered by certain changes of the processor speed(s). The correctness of the system consists of separate validations under each running mode. E.g., under the dual-criticality case, deadline meeting guarantees are made to all tasks or jobs under LO-criticality mode, while only to more important ones under HI-criticality mode.

The drop of platform performance may be observed by executions of workloads exceeding certain thresholds, hence existing work for scheduling Vestal's MC systems (with multiple WCET

specifications) can be used to schedule this transformed system, and that the resulting scheduling strategy correctly schedules the MC system under our interpretation (upon the varying-speed processor). However, in this section, we have successfully show that one can sometimes do *better* if using our MC model:

- For scheduling MC job set on uniprocessor platforms, (Baruah et al., 2012a) have shown its NP-hardness in the strong sense under the multiple-WCET model, whereas Sec 4.2 provides an optimal (linear programming based) polynomial-time algorithm for solving the same problem in our model, under the assumption that processor is aware of their execution speeds (self-monitoring). Note that this work does not restrict the number of criticality levels to be 2.

- The work described in Sec 4.2 is extended for multiprocessor platforms in Sec 4.4. To retain the optimality result, we show that one has to mimic a processor sharing scheme, and provided two optimal online strategies to transform the solution of the linear program. As described in Chapter 3, the best-known speedup for MC scheduling on multiprocessors is $(\sqrt{5}+1)/2$ before our work, and $4/3$ in Sec. 3.3. While here we provide an optimal scheduler (at a cost of numerous preemptions), which means the speedup is 1.

- We also extend the LP-based algorithm for scheduling MC task set on uniprocessor platforms. The optimality property can be retained with the proposed Preemptive-MC algorithm, under the assumption that fluid schedule (i.e., processor-sharing with an unlimited number of preemptions) is allowed.

- We further investigate the privilege of self-monitoring, by removing such self-awareness assumption in Sec. 4.3. For the non-monitored case, we are not able to propose an optimal scheduler like the LP based one in the self-monitored case. However, we found that an existing algorithm named OCBP (see 3.1.3 for a detailed description) can be adapted at no significant schedulability loss, in the sense that the speedup (over any clairvoyant algorithm) can be upper bounded by $(\sqrt{5}+1)/2$, and stays even lower when degraded speed varies.

CHAPTER 5: WHEN MC ARISES FROM MORE THAN ONE DIMENSION OF UNCERTAINTIES

Similar to the model settings described in Chapter 3, most prior work on mixed-criticality (MC) scheduling has focused on the model in which multiple WCET parameters are specified for each job. The interpretation is that the larger WCET values represent "safer" estimates of the job's true execution pattern. A different MC model has been studied in Chapter 4, where it is assumed that the precise speed of the processor upon which the system is implemented varies in an *a priori* unknown manner during runtime, and estimates must be made about how low the actual speed may fall.

In both models, the objective is to devise a scheduling strategy which ensures that (i) all jobs complete by their deadlines if the less pessimistic estimates are correct, and (ii) the more critical jobs complete correctly even if the less pessimistic estimates turn out to be incorrect (but the more pessimistic estimates remain true). More precise definitions will be given in each section separately.

The work described in this chapter seeks to integrate the varying-speed MC model and the multi-WCET one into a unified framework. To address the scheduling problem where MC arises from two dimensions, a general model is proposed in which each job may have multiple WCETs specified, *and* the precise speed of the processor upon which the system is implemented may vary during run-time. Sec. 5.1 considers workloads modeled as finite collections of jobs, while Sec. 5.2 studies the MC task scheduling problem. Throughout this chapter, we restrict the total number of criticality levels to be two: HI and LO.

5.1 Scheduling MC Job Set upon Varying-Speed Platforms

In this section, we model a mixed-criticality real-time workload as being composed of basic units of work known as mixed-criticality *jobs*. Each MC job J_i is characterized by a 4-tuple of

parameters: a release time a_i, a vector $\langle c_i^L, c_i^H \rangle$ of two WCET values where $c_i^L \leq c_i^H$ for HI-criticality jobs and $c_i^L = c_i^H$ for LO-criticality ones, a deadline d_i, and a criticality level $\chi_i \in \{\text{LO}, \text{HI}\}$.

A mixed-criticality *instance* $\mathscr{I}$ is specified by

- a collection of MC jobs: $J = \{J_1, J_2, \ldots, J_n\}$, and
- a processor that is characterized by two thresholds: a *normal* speed s_n and a *degraded* speed $s_d (\leq s_n)$.

The interpretation is that the jobs in J are to execute on a single shared preemptive processor that has two modes: a *normal* mode and a *degraded* (or *faulty*) mode. In normal mode, the processor executes as a speed-s_n (or faster) processor and hence completes at least s_n units of execution per time unit, whereas in degraded mode it completes less than s_n, but at least s_d units of execution per time unit. The processor starts out executing at or above its normal speed, and it is not *a priori* known how the processor speed will vary during run-time.

Definition 5.1. *A scheduling strategy for MC instances is* ***correct*** *if upon scheduling any MC instance $\mathscr{I} = (\{J_1, J_2, \ldots, J_n\}, s_d, s_n)$, it satisfies the following two properties P1 and P2.*

1. *Each job J_i meets its deadline if all jobs complete execution upon having executed for no more than their* LO*-criticality WCETs,* and *the processor speed remains $\geq s_n$ throughout Interval $[a_i, d_i)$; and*
2. *Each* HI*-criticality job J_i meets its deadline if all* HI*-criticality jobs complete execution upon having executed for no more than their* HI*-criticality WCETs, and the processor speed remains $\geq s_d$ throughout Interval $[a_i, d_i)$;*

A scheduling strategy for MC instances is ***partially correct*** *if it satisfies the second property above, but not necessarily the first.*

That is, a partially correct scheduling strategy ensures the correct execution of HI-criticality jobs provided the processor executes at or above its degraded speed and each HI-criticality job

completes upon executing for no more than its HI-criticality WCET. A correct scheduling strategy additionally ensures the correct execution of LO-criticality jobs if the processor executes at or above its normal speed and each job completes upon executing for no more than its LO-criticality WCET.

A *clairvoyant scheduling algorithm* is one that knows, prior to scheduling an instance, (i) precisely how much execution time each job in the instance will require in order to complete, and (ii) the precise manner in which the processor speed will vary during run-time.

Definition 5.2 (***optimal scheduling strategy***)**.** *An optimal scheduling strategy for MC instances possesses the property that if it fails to maintain correctness (partial correctness, respectively) for a given MC instance $\mathscr{I}$, then no non-clairvoyant algorithm can ensure correctness (partial correctness, resp.) for the instance $\mathscr{I}$.*

Without loss of generality, we will assume that the HI-criticality jobs in given MC instance $\mathscr{I}$ are indexed $1,2,...,n_h$ and the LO-criticality jobs are indexed $n_{h+1},...,n$. Let $t_1,t_2,...,t_{k+1}$ denote the at most $2n$ distinct values for the release time and deadline parameters of the n jobs, in strictly increasing order (redundancy is eliminated, i.e., $\forall j, t_j < t_{j+1}$). These release time and deadlines partition the time duration $[\min_i\{a_i\}, \max_i\{d_i\})$ into k *intervals*, which will be denoted as $I_1, I_2, ..., I_k$, with I_j denoting the interval $[t_j, t_{j+1})$.

Most of the contributions made in this section can be found at (Guo and Baruah, 2015a).

5.1.1 LE-EDF' — Enhanced LE-EDF

We adapt Algorithm LE-EDF proposed in Sec. 3.1.2 to schedule the MC instance in this section. Originally LE-EDF targets MC instances with multi-WCET estimations, but a constant-speed platform. Some slight modifications are necessary to address the additional dimension of uncertainty considered here. In general, to guarantee the correctness of HI-criticality jobs, we now need to be pessimistic about all possible HI-criticality behaviors including the performance drop as well.

As the modifications are rather minor, we choose not to repeat the whole algorithm here in this section. Instead, we only highlight the potential differences or changes:

- In Steps 1 and 2, the intervals and HI-criticality sub-jobs are determined by considering the jobs executing upon a speed-s_d processor, instead of speed-1.

- During run-time, the trigger for mode switch may still be certain execution (of HI-criticality job) exceeds the less pessimistic assumption, and (in addition) may also be a detection of performance drop below s_n. Note that the correctness of HI-criticality jobs can be guaranteed (from the manner in which they are defined) regardless of whether or when the processor degrades into slower speeds.

- Time complexity remains the same as $\Theta(n \log n)$, where n is the total number of jobs.

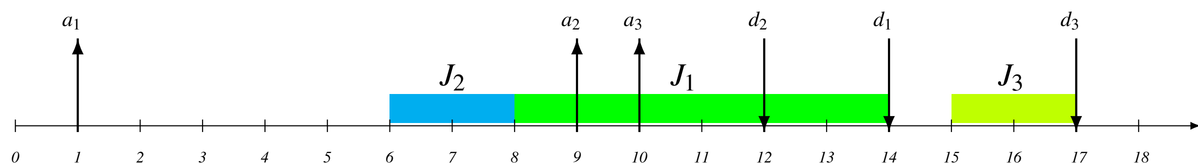
We illustrate the modified LE-EDF (named LE-EDF') in Example 5.3 below.

Example 5.3. *Throughout this section, we will consider the instance consisting of the six jobs J_1–J_6 shown in tabular form in Figure 5.1, that is to be implemented upon a preemptive processor of normal speed $s_n = 1$ and degraded speed $s_d = 0.5$.*

J_i	a_i	c_i^L	c_i^H	d_i	χ_i
J_1	1	2	3	14	HI
J_2	9	0.5	1	12	HI
J_3	10	0.5	1	17	HI
J_4	0	7	7	10	LO
J_5	1	0.5	0.5	12	LO
J_6	12	3	3	16	LO

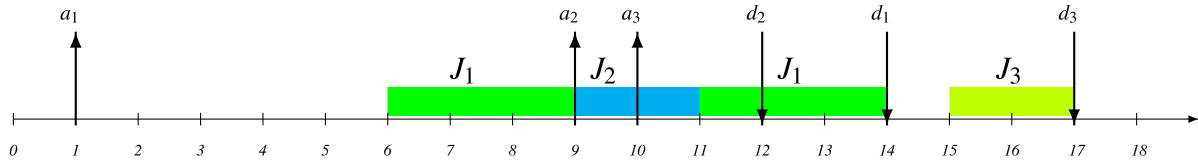
Figure 5.1: An example MC collection of jobs.

Step 1. *Consider only the HI-criticality jobs J_1–J_3 executing for their HI-criticality WCETs on a speed-s_d processor, the intervals identified in Step 1 are as follows:*



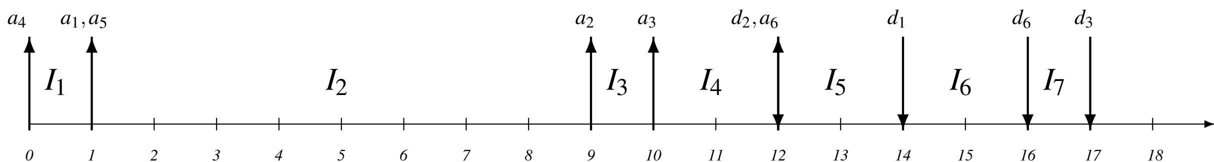
The intervals determined in Step 1 are therefore $[6,14)$ *and* $[15,17)$. *(Observe that in this schedule we are only determining execution intervals, not seeking to determine an actual schedule. Hence the fact that job* J_2 *seems to be "assigned" execution prior to its release time is irrelevant.)*

Step 2. *The EDF schedule for the* HI*-criticality jobs upon a speed-*0.5 *processor is then constructed only within the intervals identified in Step 1; i.e.,* $[6,14),[15,17)$[1]*:*



- J_1 *executes during the interval* $[6,9)$ *as the only active job.*
- *Upon release,* J_2 *becomes the earliest-deadline job and is hence allocated execution over the interval* $[9,11)$.
- *Upon* J_2*'s completion,* J_1 *executes during the interval* $[11,14)$ *as the only active job.*
- J_3 *executes in the interval* $[15,17)$ *as the only active job.*

Step 3. *The timeline is partitioned into seven intervals* $[0,1)$, $[1,9)$, $[9,10)$, $[10,12)$, $[12,14)$, $[14,16)$, *and* $[16,17)$.



Each of the HI*-criticality jobs is decomposed into the sub-jobs shown in Figure 5.2; these are obtained by super-imposing the partitions shown above upon the constructed EDF schedule.*

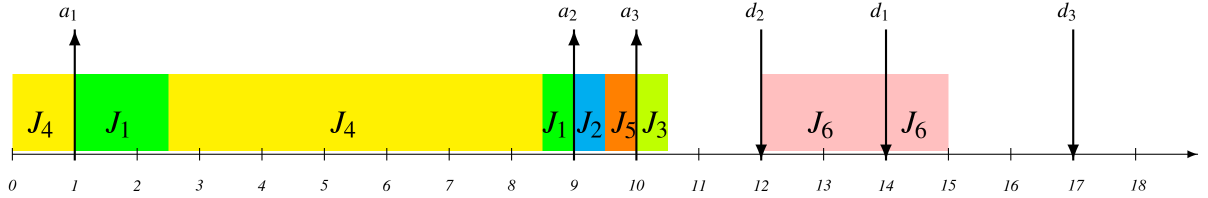
[1] Note that Step 1 may result in new breakpoints to the timeline and intervals other than release time and deadlines; e.g., $t = 6$.

J_i	a_i	c_i^H	d_i	χ_i
J_{12}	1	1.5	9	HI
J_{14}	1	0.5	12	HI
J_{15}	1	1	14	HI
J_{23}	9	0.5	10	HI
J_{24}	9	0.5	12	HI
J_{36}	10	0.5	16	HI
J_{37}	10	0.5	17	HI

Figure 5.2: HI-criticality sub-jobs generated by Step 3 of LE-EDF' in Example 5.3.

Run-Time Scheduling. *The processor speed may fall below its nominal value at any instant during execution. To better illustrate how our algorithm works, we separately demonstrate its operation under three different run-time behaviors of the system.*

§1. *We first consider the case where no degradation in processor speed occurs, and all* HI*-criticality jobs execute at their* LO*-criticality WCETs. The schedule is depicted in the following figure. (Since sub-job numbers align with interval number, we only label the job numbers.)*



- *For Interval $I_1 = [0,1)$, since no* HI*-criticality sub-job is allocated here, J_4 will be executed as the earliest deadline* LO*-criticality job.*

- *Sub-job J_{12} executes for 1.5 time units at the beginning of Interval $I_2 = [1,9)$. The remaining capacity will be used for jobs with deadline greater than 9. As the earliest deadline* LO*-criticality job, J_4 executes first and completes at $t = 8.5$, after which J_{14} executes over the interval $[8.5,9)$ (and also completes).*

- *Sub-job J_{23} is executed first in Interval $I_3 = [9,10)$, and completes at time $t = 9.5$. The earliest deadline active job (which is J_5) executes over the interval $[9.5,10)$.*

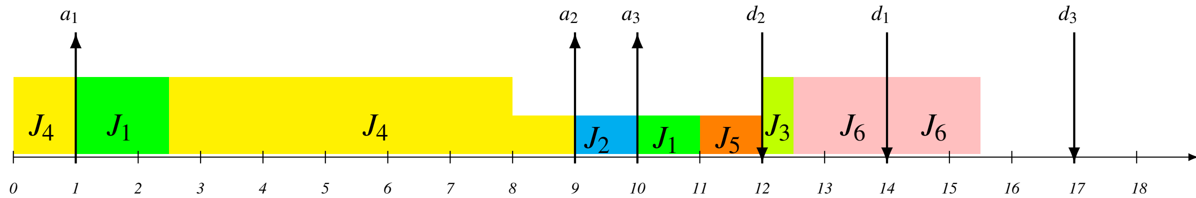
- *Since all* HI*-criticality jobs execute at their* LO*-criticality WCETs, both J_1 and J_2 are already finished at $t = 10$, and sub-jobs J_{15} and J_{24} require no execution. As a result,* HI*-criticality sub-job J_{36} (as the only active sub-job) will be executed in Interval $I_6 = [10, 10.5)$. We detect idleness throughout the rest of the interval; i.e., $[10.5, 12)$,*

- *Interval $I_5 = [12, 14)$ is empty and should be used for the only active job J_6.*

- *The only active* LO*-criticality job J_6 executes until it completes at $t = 15$. Now the processor becomes idle since J_{37} is an inactive sub-job, J_3 having already completed upon completing sub-job J_{36}.*

- *The processor idles during Interval $I_7 = [16, 17)$.*

§2. *Next, we consider another case where the processor speed degrades to 0.5 over the time-interval $[8, 12)$. We assume that all* HI*-criticality jobs execute at their* LO*-criticality WCETs (and thus sub-jobs J_{15}, J_{24}, and J_{37} can be ignored[2]).*



- *Execution in Interval $I_1 = [0, 1)$ is the same as in the previous case.*

- *Compared to the previous scenario, the amount of computing capacity available in Interval $I_2 = [1, 9)$ is less now due to the degradation of processor speed. After completing sub-job J_{12}, I_2 is only able to execute J_4, which completes at time instant 9.*

- *Interval $I_3 = [9, 10)$ also suffers from the degradation, and is fully consumed by the sub-job J_{23}.*

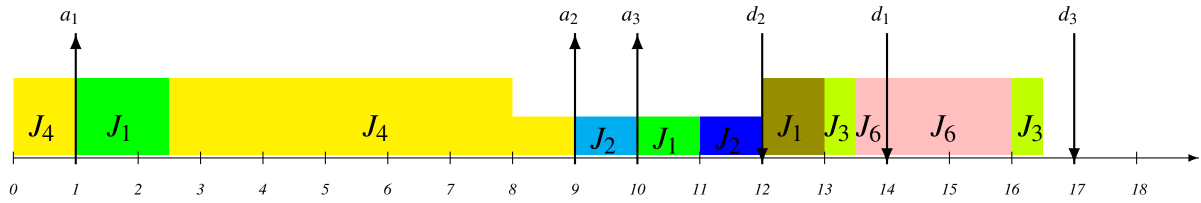
[2]Of course these sub-jobs will not actually be ignored during run-time; rather, they will be determined to be inactive (as it is explained in the case above). Here we simply ignore them in order to simplify the explanation.

- *The processor remains in degraded mode for Interval $I_4 = [10, 12)$, where* HI*-criticality sub-job J_{14} executes and completes at time instant 11. The remaining one time unit is used for executing* LO*-criticality job(s): J_5 executes from $t = 11$ to $t = 12$ and meets its deadline.*

- *The processor recovers to normal speed at time $t = 12$, and the executions in the remaining three intervals are the same as in the previous case.*

Note that although the processor operated in degraded mode for four time units, LE-EDF' *nevertheless completed all the jobs by their deadlines.*

§3. *As a final part, we consider the case where the processor suffers from a degradation between $t = 8$ and $t = 12$,* ***and*** HI*-criticality jobs J_1 and J_2 execute at their* HI*-criticality WCETs (for those reading this on a color monitor, execution beyond the* LO*-criticality WCET is depicted with darker colors).*



- *Execution in Intervals $I_1 = [0, 1)$, $I_2 = [1, 9)$, and $I_3 = [9, 10)$ remains the same as in the previous case.*

- *Both J_{14} and J_{24} need to complete within interval $I_4 = [10, 12)$. No capacity remains due to the processor degradation, and the unfinished* LO*-criticality job J_5 is dropped at its deadline $t = 12$.*

- *At the beginning of Interval $I_5 = [12, 14)$, the processor recovers to normal speed. The interval $[12, 13)$ is consumed by J_{15}. At time $t = 13$, there are two active jobs J_{36} and J_6 with the same deadline, and according to the algorithm, we favor* HI*-criticality jobs in such case, which results in the execution of J_3 within $[13, 13.5)$, and then J_6 afterward.*

- *There are two active jobs (J_{37} and J_6) within Interval $I_6 = [14, 16)$. J_6 executes first since it has got an earlier deadline (although with lower criticality level). Unfortunately, J_6 may be dropped at its deadline $t = 16$ since it has only received $2\frac{1}{2}$ units of execution (which is fewer than the required three units).*

- *Sub-job J_{37} executes in Interval $I_7 = [16, 17)$, completing at time instant 16.5. The processor is idled for the remainder of the interval.*

It is instructive to review the last scenario considered in the example above, where two LO-*criticality jobs J_5 and J_6 miss their deadlines.*

The situation for J_5 within Interval $I_4 = [10, 12)$ is straightforward — the processor is suffering from a degradation within this interval, and since J_1 and J_2 are both HI-*criticality jobs, the sub-jobs J_{14} and J_{24} certainly need to be prioritized over the* LO-*criticality job J_5.*

The argument for J_6 to miss its deadline is not quite as unequivocal: a scheduling algorithm that postponed the execution of sub-job J_{36} to interval I_7 (where, as we saw, there is adequate excess capacity to accommodate this sub-job) and instead executed J_6 for an additional one-half unit during interval I_6 would have seen both J_6 and J_3 complete by their deadlines. However, such a scheduling algorithm would need to know beforehand (i.e., during executing in Interval I_6) that the processor speed would not degrade during interval I_7. That is, such an algorithm would need to be clairvoyant*.*

5.1.2 Online Optimality Under Single WCET Case

The failure of LE-EDF' to correctly schedule an instance that would be scheduled correctly by a clairvoyant algorithm does not rule out the possibility that LE-EDF' is an optimal algorithm: according to Definition 4.20, an optimal scheduling strategy should be able to correctly schedule any instance that can be correctly scheduled by a non-clairvoyant scheduling strategy.

In this subsection, we show the optimality of LE-EDF' under single WCET case, i.e., for each job J_i it is the case that $c_i^L = c_i^H$. Note that for this case, an LP based optimal scheduler has

already been proposed in Sec 4.2. Since (as we saw above) LE-EDF' can be implemented to have a run-time that is $\Theta(n \log n)$ for an instance composed of n jobs while LP-solvers have significantly poorer (although still polynomial) run-times, we argue that LE-EDF' is a preferred algorithm for scheduling such instances.

Lemma 5.4. *If a* LO*-criticality job J_i with release time a_i and deadline d_i is dropped by* LE-EDF' *during run-time, the processor remains busy in the interval $[a_i, d_i)$. Furthermore, no* HI*-criticality execution that had been allocated to later intervals (than d_i) in the pre-computed scheduling table gets executed within this interval.*

Proof: It is easy to see that job J_i remains *active* (released and unfinished) throughout this whole interval. Thus, there must be no idleness. Since our algorithm only "promotes" pre-allocated HI-criticality amounts when the processor idles, we know that no HI-criticality amount can be transferred from later intervals into $[a_i, d_i)$. □

Theorem 5.5. LE-EDF' *is an* optimal scheduling strategy *for MC instances in which $c_i^L = c_i^H$ for all jobs J_i.*

Proof: From the definition of an optimal scheduling strategy (Definition 5.2), it follows that we have two proof obligations here.

First, we must show that LE-EDF' is able to schedule in a partially correct manner any instance that can be scheduled in a partially correct manner by any non-clairvoyant algorithm. Partial correctness trivially follows from the optimality of EDF: if any non-clairvoyant algorithm is able to satisfy the second property of Definition 5.1, it follows from the manner in which we construct the scheduling table in Steps 1 and 2 of Sec. 3.1.2.1 that LE-EDF' will also satisfy the second property.

Second, we must show that LE-EDF' is able to correctly schedule any instance that can be correctly scheduled by any non-clairvoyant algorithm. Suppose that both LE-EDF' and some other (non-clairvoyant) algorithm are both able to schedule a given MC instance $\mathscr{I}$ in a partially correct manner, but LE-EDF' is unable to correctly schedule $\mathscr{I}$ — it drops a LO-criticality job J^* during run-time. Let a^* denote the release time, and d^* the deadline, of this job J^*. We argue that

any non-clairvoyant scheduler that completes all HI-criticality jobs (and thereby satisfies partial correctness) must also fail to meet the deadline of J^* or some other LO-criticality job with a deadline at or prior to time instant d^*. This is because, in order to ensure partial correctness in the event of the processor speed degrading to s_d at some future point in time, a non-clairvoyant scheduler must make the most conservative assumptions regarding the future speed of the processor and assume that the speed will, indeed, fall to s_d. But LE-EDF' also makes this assumption, and ensures that under this assumption, the *minimum* possible amount of execution of HI-criticality jobs with deadline greater than d^* has occurred within the interval of interest. According to Lemma 5.4, no HI-criticality sub-job with deadline greater than d^* will be executed within $[a^*, d^*)$, since J^*, with an earlier deadline, is prioritized by LE-EDF'. This implies that the maximum possible amount of execution to LO-criticality jobs have occurred in the LE-EDF' schedule prior to d^*; the fact that LE-EDF' is forced to nevertheless drop a job at d^* implies that the processor is overloaded prior to d^* (and hence no other algorithm can complete all LO-criticality jobs prior to d^*). □

5.1.3 The Speedup of Non-Clairvoyance

In addition to proving the optimality of Algorithm LE-EDF' for scheduling such MC instances, we use the speedup factor metric to quantify the cost of non-clairvoyance. Speedup here is the smallest multiplicative factor (to the execution speed) LE-EDF' would need to schedule any instance that can be scheduled by a (hypothetical) clairvoyant algorithm.

Theorem 5.5 above shows that LE-EDF' is an optimal algorithm for scheduling MC instances in which each job's LO-criticality WCET is equal to its HI-criticality WCET,in the sense that no non-clairvoyant scheduler can guarantee correctness (partial correctness, respectively) if LE-EDF' is unable to do so. Note that the proof of Theorem 5.5 fundamentally depends on the fact that the algorithm being compared to is non-clairvoyant: a non-clairvoyant algorithm must necessarily assume at each instant during run-time that in the future the processor will execute throughout at its minimum (degraded) speed of s_d. In contrast, a clairvoyant algorithm may know how the processor speed will vary in the future; such an algorithm will generally outperform LE-EDF' since LE-EDF'

sometimes drops LO-criticality job to prevent future deadline misses by HI-criticality jobs due to possible processor degradation that may not happen. The third scenario considered in Example 5.3 had illustrated that a clairvoyant algorithm may ensure correctness while LE-EDF' is only partially correct.

In this section, we will quantify the gap between LE-EDF' and any optimal clairvoyant algorithm using the metric of *speedup factor* (Kalyanasundaram and Pruhs, 2000). The use of this metric for the purposes of quantifying the cost of non-clairvoyance seems particularly appropriate: the seminal paper (Kalyanasundaram and Pruhs, 2000) on speed factors was titled "*Speed is as powerful as clairvoyance*," which is what we, too, establish in this section (albeit for a completely different problem than the one considered in (Kalyanasundaram and Pruhs, 2000)).

According to the load definition in Def. **??**, it is easily seen that a necessary and sufficient condition for an optimal clairvoyant algorithm to successfully schedule MC instance $\mathscr{I} = (J, s_n, s_d)$ is that $\ell_{\text{LO}}(J) \leq s_n$, and $\ell_{\text{HI}}(J) \leq s_d$. A natural question arises: can we determine a speedup factor $s(> 1)$ for Algorithm LE-EDF' such that a sufficient condition for LE-EDF' to schedule MC instance $\mathscr{I} = (J, s_n, s_d)$ in a correct manner (see Definition 3.1) is that $\ell_{\text{LO}}(J) \leq s \times s_n$, and $\ell_{\text{HI}}(J) \leq s \times s_d$? The following theorem leads us to an answer:

Theorem 5.6. *If an MC instance $\mathscr{I} = (J, s\ell_{\text{LO}}(J), s\ell_{\text{HI}}(J))$ that is schedulable by an optimal clairvoyant algorithm is not correctly scheduled by* LE-EDF'*, then*

$$s < \frac{1}{1 - \ell_{\text{HI}}(J) + \ell_{\text{HI}}^2(J)/\ell_{\text{LO}}(J)}. \tag{5.1}$$

Proof: (of Theorem 5.6). It is evident from the manner in which the scheduling table is constructed by Algorithm LE-EDF' (in Steps 1–3) that a degraded speed of $\ell_{\text{HI}}(J)$ is already sufficient to have HI-criticality jobs meet their deadlines. It is straightforward to observe that LE-EDF' is *sustainable* (Baruah and Burns, 2006) with respect to processor speed (i.e., a faster processor would only reduce the execution time cost, and contribute positively its schedulability). Hence, LE-EDF' remains correct if provided a faster processor which executes at degraded-speed of $s\ell_{\text{HI}}(J)$. As a

result, if LE-EDF' fails to maintain correctness for a given MC instance $\mathscr{I} = (J, s\ell_{\text{LO}}(J), s\ell_{\text{HI}}(J))$, for any $s \geq 1$, the only possibility is that a LO-criticality job J_i is dropped at its deadline d_i — we study this only possible scenario in the following to derive a bound on the speedup factor s.

Based on Lemma 5.4, consider any interval $[a, d)$ which contains $[a_i, d_i)$; i.e., $a \leq a_i$ and $d_i \leq d$. Since we dropped a LO-criticality job at time $t = d_i$, the most pessimistic assumption is that our processor runs at degraded speed $s\ell_{\text{HI}}(J_{\text{HI}})$ thereafter, and moreover we fully utilize interval $[d_i, d)$ with HI-criticality jobs. When compared to the clairvoyant execution of such a job set, the only difference for the interval $[a, d_i)$ is that the additional capacity from the speedup $s(d_i - a_i)$ may be used to execute HI-criticality jobs with further deadlines. However, those HI-criticality jobs at the same time suffer from the degradation after time $t = d_i$, such that the provided capacity is not enough to guarantee them meeting deadlines. This is exactly the reason why our algorithm will pre-allocate more HI-criticality amounts into the interval $[a, d_i)$, and thus cause the job J_i miss its deadline. Intuitively speaking, the additional capacity provided within the interval $[a, d_i)$ is not enough to cover the "needs" from HI-criticality jobs that are executed later in the interval $[d_i, d)$ by the clairvoyant algorithm. Thus, the following inequality must hold for any $a \leq a_i$, in order for LE-EDF' to drop LO-criticality job J_i at its deadline.

$$\big(s\ell_{\text{LO}}(J) - \ell_{\text{LO}}(J)\big)(d_i - a) < \big(\ell_{\text{LO}}(J) - s\ell_{\text{HI}}(J)\big)(d - d_i) \tag{5.2}$$

The worst case is obtained by setting $a = a_i$, and this yields an upper bound on s. Without loss of generality, we assume $d - a_i = 1$, and denote $x := d - d_i \in [0, \ell_{\text{HI}}(J)]$. Since we only consider *active* HI-criticality jobs within the interval, x cannot exceed $\ell_{\text{HI}}(J)$ or else not even a clairvoyant algorithm would finish them on time. Inequality (5.2) can be re-written in the following manner with respect to the speedup factor s:

$$\forall x \in [0, \ell_{\text{HI}}(J)], s < \frac{1}{1 - x + x\frac{\ell_{\text{HI}}(J)}{\ell_{\text{LO}}(J)}} \tag{5.3}$$

When $\ell_{\text{HI}}(J) \geq \ell_{\text{LO}}(J)$, we simply have $s < 1$ which is not the case of interest. When $\ell_{\text{HI}}(J) < \ell_{\text{LO}}(J)$, the right-hand side of 5.3 is monotonically increasing with respect to x, the upper bound of the speedup factor becomes tight when x takes its largest possible value $\ell_{\text{HI}}(J)$, which will lead us to:

$$s < \frac{1}{1 - \ell_{\text{HI}}(J) + \ell_{\text{HI}}^2(J)/\ell_{\text{LO}}(J)}. \tag{5.4}$$

and the theorem follows. □

Analysis of Inequality (5.1) yields the following corollary.

Corollary 5.7. *The upper bound of the speedup factor is $s_{\max} = 4/3$, which occurs when $\ell_{\text{LO}}(J) = 1$ and $\ell_{\text{HI}}(J) = 0.5$.* □

Proof: The result comes from two simple facts: (i) the right hand side of Inequality (5.4) monotonically increases as $\ell_{\text{LO}}(J)$ increases; (ii) $1 - x + x^2 \geq 3/4$, and $= 3/4$ only when $x = 1/2$. □

5.2 Scheduling MC Task Set upon Varying-Speed Platforms

In this section, we seek to integrate both these dimensions of uncertainties for MC systems composed of recurrent tasks. The techniques that need to be developed, and the results obtained, are strikingly different than the independent job case studied in the previous section. Most of the contributions made in this section can be found at (Baruah and Guo, 2014).

5.2.1 Model and Definitions

An **MC instance** $\mathscr{I}$ is specified as a finite collection of MC tasks τ and a varying-speed processor characterized by a degraded speed s (and normal speed of 1). [3]

[3] Assuming the readers are now quite familiar with MC task set models, we directly introduce the system behavior and correctness criterion.

System behavior. During execution, the system exhibits LO-criticality behavior if (i) each job $\tau_{i,j}$ (released by task τ_i) signals completion without exceeding C_i^L time units of execution, and (ii) the execution speed of the processor never falls below 1. The system is in HI-criticality behavior if platform execution speed falls below 1 but no lower than s, or some job $\tau_{i,j}$ did not signal finishing when exhausted its C_i^L, but no greater than C_i^H/s. Otherwise, it exhibits erroneous conditions, which is not of our interest.

Correctness criterion. We define an algorithm for scheduling MC instances to be *correct* if it is able to schedule any system such that

- During all LO-criticality behaviors of the system in which the processor speed remains at or above 1, all jobs receive enough execution between their release time and deadline to signal completion; and
- During all HI-criticality behaviors of the system, all HI-criticality jobs receive enough execution between their release time and deadlines to signal completion provided the processor speed remains at or above s.

The correctness definition is quite similar to Def. 5.1. That is, if the system exhibits LO-criticality behavior and the processor exhibits normal behavior, then all deadlines should be met; else, all HI-deadlines should be met (provided neither the system nor the processor exhibits erroneous behavior).

Note that if any job executes for more than its LO-criticality WCET or the processor speed falls below 1, we do not require any LO-criticality jobs (including those that may have arrived before this happened) to complete by their deadlines. This is a consequence of the nature of system validation: informally speaking, the system designer fully expects that the system will exhibit LO-criticality behavior and the processor always execute at or above its normal speed, and hence is only concerned that they behave as desired under these circumstances. The validation process for the more critical functionalities, on the other hand, allows for the possibility that some jobs may exhibit HI-criticality behavior and/ or the processor executes at a speed slower than 1 (but $\geq s$), and requires that all

HI-criticality jobs nevertheless meet their deadlines; however, such validation is not concerned with the fate of the LO-criticality jobs.

A clairvoyant scheduling algorithm is one that knows, prior to scheduling an instance, (i) precisely how much execution each job in the instance will require in order to complete, and (ii) the precise manner in which the processor speed will vary during run-time.

5.2.2 Non-Monitoring Processors

We propose an algorithm VDF-NM (for Virtual-Deadline First Non-Monitoring) for scheduling systems that do not possess the capability of knowing its speed at each instant in time. VDF-NM is motivated by, and hence quite similar to, the EDF-VD algorithm that was proposed in (Baruah et al., 2012b).

Overview. Prior to run-time, VDF-NM performs a schedulability test to determine whether the given set τ can be successfully scheduled by it or not. If τ is deemed schedulable, then an additional parameter, which we call a modified period denoted $\hat{T}_i$, is computed for each HI-criticality task $\tau_i \in \tau$. The algorithm for computing these parameters is described in the pseudo-code form in Figure 5.3, with correctness proved in Theorems 5.8 and 5.9. Run-time scheduling is done according to the EDF order of modified deadlines.

During run-time, if some job executes for a duration exceeding its LO-criticality WCET without signaling that it has completed execution, we know that the system is no longer exhibiting LO-criticality behavior. In response, the run-time scheduler immediately discards all LO-criticality jobs; subsequently, only HI-criticality jobs will receive further execution. Subsequent execution of HI-criticality tasks (including the jobs that are currently active) continue to be done according to EDF. But the actual job deadlines (arrival time plus period) are used.

Theorem 5.8. *The following condition is sufficient for ensuring that VDF-NM successfully schedules all* LO*-criticality behaviors of* τ*:*

$$x \geq \frac{U_H^L}{1 - U_L^L}. \tag{5.6}$$

Given MC instance (τ, s)

1) Compute x as follows: $x \leftarrow \frac{U_H^L(\tau)}{1-U_L^L}$;

2) **If** $U_H^H/(1-x) \le s$, **then**

For each HI-criticality task τ_i;

$$\hat{T}_i \leftarrow xT_i; \tag{5.5}$$

Return success;

Else Return failure;

Figure 5.3: VDF-NM: The preprocessing phase.

Proof: If EDF is able to schedule, upon a unit-speed processor, all LO-criticality behaviors of the task system obtained from τ by replacing each HI-criticality task τ_i by one with a reduced period, then it follows from the sustainability (Baruah and Burns, 2006) of uniprocessor EDF that EDF is able to schedule all LO-criticality behaviors of τ upon a unit-speed processor as well. Note that scaling down the period of each HI-criticality task by a factor x is equivalent to inflating its utilization by a factor 1/x. Since the utilization bound of EDF for implicit-deadline tasks is known to be equal to the processor capacity (see Theorem 2.3), we therefore conclude that

$$\left(U_L^L + \frac{U_H^L}{x} \le 1\right) \Leftrightarrow \left(x \ge \frac{U_H^L}{1-U_L^L}\right).$$

is sufficient for ensuring that VDF-NM successfully schedules all LO-criticality behaviors of τ. □

Theorem 5.9. *The following condition is sufficient for ensuring that VDF-NM successfully schedules all* HI-*criticality behaviors of* τ*:*

$$s \ge \frac{U_H^H}{1-x}. \tag{5.7}$$

Proof: Suppose that at some instant t^* during run-time, the scheduler detects that some job has executed for a duration exceeding its LO-criticality WCET without signaling completion. It

immediately discards all LO-criticality jobs, re-assigns each active HI-criticality job a deadline equal to its release time plus the original period of the task that generated it, and assigns each future-arriving HI-criticality job a deadline equal to its release time plus the period of the task that generates it.

Since the modified relative deadline of a job of HI-criticality task τ_i is equal to xT_i, if this job is active at time instant t^* its actual deadline must be at least $(T_i - xT_i)$ time units in the future. The utilization of task τ_i beyond time instant t^* is therefore no greater than that of an implicit-deadline sporadic task with execution requirement C_i^H and period $(T_i - xT_i)$. Summing over all HI-criticality tasks and using once again the fact that EDF has a utilization bound equal to the processor capacity, we conclude that

$$\sum_{\chi_i = \text{HI}} \frac{C_i^H}{T_i - xT_i} \Leftrightarrow \frac{U_H^H}{1-x}.$$

is a sufficient condition for VDF-NM to meet all HI-criticality job deadlines upon the degraded processor of speed $\geq s$. □

The top-level idea behind Algorithm VDF-NM is essentially this: determine the smallest scaling factor $x < 1$ such that the system with HI-criticality deadlines scaled by a factor x remains EDF-schedulable in LO-criticality behaviors, and then determine whether shrinking HI-criticality deadlines in this manner will allow all HI-criticality deadlines to be guaranteed meet in the event of a HI-criticality behavior being identified (see Figure 2 above). Both the LO-criticality and the HI-criticality schedulability testing is done via the utilization-based EDF schedulability test. For the LO-criticality schedulability testing, each HI-criticality task τ_i is modeled as a task with WCET C_i^L and period xT_i; for HI-criticality schedulability testing, it is modeled as a task with WCET C_i^H and period $(1x)T_i$.

Although this approach is correct (according to Theorems 5.8 and 5.9), it can be pessimistic as the conditions are sufficient only.

A pragmatic improvement. The algorithm we advocate in the remainder of this subsection, VDF-NM+, takes the following approach to reduce pessimism: for the purposes of doing the schedulability

analyses, model each HI-criticality task τ_i as constrained-deadline (rather than implicit-deadline) tasks by:

- For LO-criticality schedulability analysis, model it as a constrained-deadline task with WCET C_i^L, relative deadline xT_i, and period T_i;
- For HI-criticality schedulability analysis, model it as a constrained-deadline task with the parameters WCET C_i^L, relative deadline $(1x)T_i$, and period T_i.

Although EDF-schedulability analysis of constrained deadline sporadic task systems is NP-hard (F. Eisenbrand and T. Rothvoß, 2010), polynomial time approximation schemes (PTASs) are known (see, e.g., (Albers and Slomka, 2004)) that can solve this problem in efficient polynomial time to any desired degree of accuracy. We have therefore implemented the following method for computing the scaling factor x that is used by VDF-NM:

- Use binary search over the range $(0,1)$ to determine, to any desired degree of accuracy, the smallest value of x for which the constrained-deadline task system:
  $\cup_{\chi_i=\text{LO}}\{(C_i^L,T_i,T_i)\}\bigcup\cup_{\chi_i=\text{HI}}\{(C_i^L,xT_i,T_i)\}$
  is EDF-schedulable.
- For the value of x determined above, check whether the constrained-deadline task system $\cup_{\chi_i=\text{LO}}\{(C_i^H,(1-x)T_i,T_i)\}$ is EDF-schedulable. If so, use this value of x as the scaling factor in Step 2 of Fig. 5.3; else, declare failure.

This is clearly a strict improvement over the method for computing the scaling factor used in Step 1 of Fig. 5.3, in the sense that the value of x computed can only be smaller, and hence failure will be declared for fewer systems. Experimental study will be reported in Sec. 5.2.4, which validates our theoretical analysis.

5.2.3 Self-Monitoring Processors

We now consider the case where the processor is aware of its execution speed at any instant during run-time. We define an algorithm, VDF-WM (for Virtual-Deadline First - With Monitoring),

that may trigger a mode switch when some job executes for a duration exceeding its LO-criticality WCET without signaling that it has completed execution (as with VDF-NM), *or the processor speed is observed to fall below its normal value of* 1.

The pre-runtime processing phase (Step 1 in Figure 5.3) for VDF-WM is identical to VDF-NM — the same scaling factor $x = U_H^L(\tau)/(1 - U_L^L(\tau))$ is computed. However, the acceptance test (i.e., Step 2 of the pseudo-code) is different: VDF-WM checks to determine whether the value of x computed in Step 1 satisfies:

$$xU_L^L + U_H^H \leq s. \tag{5.8}$$

Since the scaling factor x used by VDF-WM is the same as the one used by VDF-NM, Theorem 5.8 continues to hold and VDF-WM is therefore seen to schedule all LO-criticality behaviors correctly. In Theorem 5.10 below, we prove that all HI-criticality behaviors are also scheduled correctly:

Theorem 5.10. *The condition listed in (5.8) is sufficient for ensuring that VDF-WM successfully schedules all* HI*-criticality behaviors of* τ.

Proof: Suppose that VDF-WM cannot meet all deadlines in all HI-criticality behaviors of τ. Let I denote a minimal instance of jobs released by τ, on which a deadline is missed. Without loss of generality, assume that the earliest job-release in I occurs at time zero, and let t_f denote the instant of the (first) deadline miss since (as argued above) Theorem 5.8 holds for VDF-WM, this must be the deadline of a HI-criticality job, in a HI-criticality behavior. Let t^* denote the time instant at which HI-criticality behavior is first flagged (i.e., the first instant at which some job executes for more than its LO-criticality worst-case execution time without signaling that it has completed execution).

Some notations:

- For each $i, 1 \leq i \leq n$, let η_i denote the amount of execution over the interval $[0, t_f]$ that is needed by jobs in I that are generated by task τ_i.

- For each $i, 1 \leq i \leq n$, let $u_i(\chi)$ denote the per criticality level utilization C_i^χ / T_i.
- Let J_1 denote the job with the earliest release time amongst all those that execute in $[t^*, t_f)$. Let a_1 denote its release time, and d_1 its deadline. (Note that $a_1 \leq t^*$.)

Lemma 5.11. *All jobs that execute in $[t^*, t_f)$ have deadline $\leq t_f$.*

Proof: Suppose not. Consider the latest instant t' in $[t^*, t_f)$ when a job with deadline $> t_f$ executes. Only those jobs in I that have release time $\geq t'$ and deadline $\leq t_f$ are sufficient to cause a deadline miss; this contradicts the assumed minimality of I. □

It immediately follows that $d_1 \leq t_f$.

Lemma 5.12.

$$\forall i, \chi_i = \text{LO}, \eta_i \leq u_i^L (a_1 + x(t_f - a_1)). \tag{5.9}$$

Proof: No LO-criticality job will execute after t^*. For it to execute after a_1, it must have a deadline no larger than J_1's virtual deadline, which is $a_1 + x(d_1 a_1)$. Therefore, no LO-criticality job with deadline $> a_1 + x(d_1 a_1)$ will execute after a_1.

Suppose that some LO-criticality job with deadline $> a_1 + x(d_1 a_1)$ were to execute, at some time $< a_1$. Let t' denote the latest instant at which any such job executes. This means that at this instant, there were no jobs with effective deadline $\leq a_1 + x(t_f a_1)$ awaiting execution. Hence by considering only those jobs in I that have release times $\geq t'$, the instance (with this LO-criticality task removed) also misses a deadline; this contradicts the assumed minimality of I. □

Lemma 5.13.

$$\forall i, \chi_i = \text{HI}, \eta_i \leq \frac{u_i^L}{x} a_1 + (t_f - a_1) u_i^H. \tag{5.10}$$

Proof: We consider separately the cases when τ_i does not have a job with release time $\geq a_1$, and when it does.

Case A: If τ_i does not release a job at or after a_1. We claim that each job of τ_i has a virtual deadline $\leq a_1 + x(t_f - a_1)$. To see why this is so, consider some job with a virtual deadline $> a_1 + x(t_f - a_1)$,

and let t' denote the latest instant at which this job executes. All jobs in I that have release times $\geq t'$ also miss a deadline; this contradicts the assumed minimality of I.

Since each job has a virtual deadline $\leq a_1 + x(t_f - a_1)$, their actual deadlines are all $\leq a_1/x + (t_f - a_1)$. Therefore, their cumulative execution requirement is at most

$$\frac{a_1}{x}u_i^L + (t_f - a_1)u_i^L \leq \frac{a_1}{x}u_i^L + (t_f - a_1)u_i^H.$$

Case B: If τ_i releases a job after a_1. Let a_i denote the first release $\geq a_1$. The cumulative execution requirement of all jobs of $_i$ is at most (since $a_1 \leq a_i, u_i^L \leq u_i^H$, and $x < 1$)

$$a_i u_i^L + (t_f - a_i)u_i^H \leq \frac{a_1}{x}u_i^L + (t_f - a_1)u_i^H.$$

□

Summing the cumulative demand of all the tasks over $[0, t_f)$ gives us:

$$\begin{aligned}
& \sum_{\chi_i = \text{LO}} \eta_i + \sum_{\chi_i = \text{HI}} \eta_i \\
\leq \quad & \sum_{\chi_i = \text{LO}} u_i^L(a_1 + x(t_f - a_1)) + \sum_{\chi_i = \text{HI}} \frac{a_1}{x}u_i^L + (t_f - a_1)u_i^H) \\
= \quad & a_1(U_L^L(\tau) + \frac{U_H^L(\tau)}{x}) + (t_f - a_1)(xU_L^L(\tau) + U_H^H(\tau)) \\
\leq \quad & a_1 + (t_f - a_1)(xU_L^L(\tau) + U_H^H(\tau)) \text{ (By(5.6))}
\end{aligned}$$

Since the amount of computation available on the processor is $t^* + s(t_f - t^*)$ and $a_1 \leq t^*$, it follows from the infeasibility of this instance that

$$\begin{aligned}
& a_1 + (t_f - a_1)(xU_L^L(\tau) + U_H^H(\tau)) > a_1 + s(t_f - a_1) \\
\Leftrightarrow \quad & (t_f - a_1)(xU_L^L(\tau) + U_H^H(\tau)) > s(t_f - a_1) \\
\Leftrightarrow \quad & xU_L^L(\tau) + U_H^H(\tau) > s.
\end{aligned}$$

Taking the contrapositive, it follows that $xU_L^L(\tau) + U_H^H(\tau) \leq s$ is sufficient to ensure HI-criticality schedulability by VDF-NM, as is claimed in this theorem. □

5.2.4 Experimental Evaluation

We have conducted a series of schedulability experiments to evaluate the relative effectiveness of the three scheduling strategies VDF-NM, VDF-NM with the pragmatic improvement (henceforth referred to as VDF-NM+), and VDF-WM in guaranteeing to correctly schedule MC implicit-deadline sporadic task systems. Our experiments were conducted upon randomly-generated task systems with generator described in Sec. 3.2.5.1.

Figure 5.4 depicts the outcome when setting parameters as follows (see Sec. 3.2.5.1 for detailed descriptions): $[U_L, U_U] = [0.02, 0.2]; [T_L, T_U] = [5, 50]; [Z_L, Z_U] = [1, 4]; P = 0.5, s = 0.8.$ The fraction of systems that were determined to be schedulable is depicted on the y-axis as a percentage, and the system utilization U_{bound} on the x-axis. Each data-point was obtained by randomly generating 1000 task systems, testing each for schedulability according to all three algorithms, and calculating the percentage of systems deemed schedulable by each algorithm.

Although we do not claim that our experiments are comprehensive enough in coverage to enable us to draw authoritative conclusions, they do point to some pretty convincing trends. It was very evident in all our experiments that VDF-NM+ consistently exhibits noticeably superior performance over VDF-NM; i.e., the pragmatic improvement to the EDF-schedulability test of VDF-NM that was described in Sec. 5.2.2 seems to provide significant benefit. Also, VDF-WM consistently exhibits noticeable improvement over VDF-NM+, indicating that self-monitoring in processors, if available, can be exploited to ensure considerable enhancement of schedulability. We do not feel comfortable making quantitative claims about the degree of such improvement based on our experiments since this is necessarily influenced by the nature of our random workload generator, but instead simply report our observations.

The percentage of schedulable systems falls off sooner, and more rapidly, for VDF-NM than for VDF-NM+, which in turn falls off more rapidly than for VDF-WM. Across all the

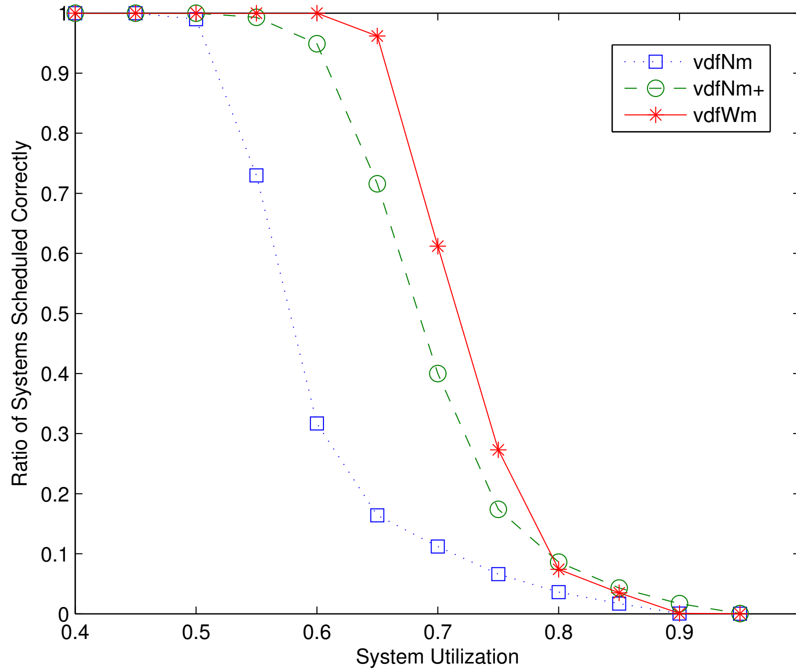


Figure 5.4: Example outcome of schedulability experiments, for parameters $[U_L, U_U] = [0.02, 0.2]; [T_L, T_U] = [5, 50]; [Z_L, Z_U] = [1, 4]; P = 0.5, s = 0.8$. The lowest line represents VDF-NM, the middle line represents VDF-NM+, and the top line represents VDF-WM.

simulation experiments that we conducted across a wide range of parameters, it appears that the simple pragmatic improvement to VDF-NM's schedulability testing that was implemented in VDF-NM+ provides between one-half to two-thirds the improvement that the more powerful platform capabilities of self-monitoring exploited in VDF-WM provides, with larger improvement ratios occurring at smaller system utilizations.

5.3 Summary

This chapter generalizes the situations and interpretations considered in previous sections. Integrated models are proposed for representing MC systems that uncertainties arises from both the WCET estimations and the platform's execution speed. The work presented in this chapter do apply to the sub-cases considered in previous parts of the dissertation. We considered both MC job set

and MC task set under the dual-criticality case, and made the following observations that support our central thesis:

- For MC job set scheduling, our newly developed LE-EDF retains the online-optimality result as the LP-based method proposed in Sec. 4.2 for single WCET case, while being computationally more efficient due to its asymptotically optimal complexity.
- Speedup study further suggests that a processor with LE-EDF scheduler is no worse than a clairvoyant processor that is $3/4$ as fast. That is, one loses no more than 25 percent of computing resource for being non-clairvoyant with LE-EDF. This is so far the best speedup result for MC job scheduling.
- We adapted the existing virtual-deadline based EDF algorithm for MC task set scheduling. We separately considered the situations where the processor is self-monitoring or not, and proposed three algorithms (that are similar to EDF-VD), which are evaluated both theoretically and experimentally via randomly generated workloads.

CHAPTER 6: CONCLUSION

Scheduling theory is applied to the analysis of *models* of systems, rather than to the physical systems themselves. In order to have confidence that the conclusions drawn on the basis of the analysis of such models will hold for the actual systems being modeled, the modeling process typically incorporates considerable pessimism into the model; such pessimism gets reflected during run-time in the form of under-utilization of platform resources that were provisioned on the basis of the pessimistic models.

Mixed-criticality (MC) scheduling theory seeks to deal with such pessimism by constructing multiple different models of a single system, and using more pessimistic models for validating the correctness of more critical functionalities whose correctness must be validated to a higher level of assurance. Prior work in MC scheduling has mostly focused on dealing with uncertainties in estimating the *upper bounds on the WCET* of pieces of code. In this dissertation, we start to study the MC scheduling problem along the dimension of varying processor speed. We have considered these two dimensions each separately, and both within a single integrated framework.

In this chapter, we first summarize the main technical contributions made in the dissertation, and then briefly introduce some other contributions made during my Ph.D. study, and point out some future research directions in the end.

6.1 Summary of Results

When MC arises solely from WCET estimations (which is Vestal's interpretation), we show that improvements to existing theories can be made via proposing new scheduler, new models, and proving better analytical results for existing schedulers. Specifically, an algorithm named LE-EDF

is proposed for scheduling MC job set, which is computationally more efficient than the well-known OCBP algorithm. We further prove that LE-EDF strictly dominates OCBP, verify such relationship by experimental study. We added one more parameter to the Vestal model, capturing the probability information about the uncertainties in system behaviors, and proposed outperforming schedulers under this more rich workload model. We also improve the speedup bound from $(\sqrt{5}+1)/2$ to $4/3$ for an existing algorithm named MC-Fluid for MC task scheduling upon the multiprocessor platform, and show that the problem is closed in the sense that no better speedup can be achieved (due to the NP-hardness nature of the problem under non-clairvoyance).

When MC arises solely from varying-speed platforms, we proposed a new model where a single WCET threshold will be assigned to each single piece of code, yet its actual run-time is related to the performance of the platform. Although a slower speed can be modeled as longer WCET, and thus existing work for scheduling Vestal's MC systems (with multiple WCET specifications) can be used to schedule this transformed system, we show that one can sometimes do *better* if using our varying-speed MC model. This is in general due to the non-NP-hardness nature under the new interpretation. Specifically, we proposed a (LP based) polynomial-time algorithm for scheduling MC jobs upon a self-monitoring uniprocessor. By mimicking processor sharing scheme (with a large number of preemptions), this work is further extended to scheduling (i) MC job set upon multiprocessor platforms and (ii) MC task set upon uniprocessor platform. When self-monitoring is not allowed, we find that the existing OCBP algorithm can be adapted at no significant schedulability loss, in the sense that the speedup can be upper bounded by $(\sqrt{5}+1)/2$, and stays even lower when degraded speed threshold varies.

We then propose integrated system models for representing MC systems that uncertainties arises from both the WCET estimations and the platform's execution speed. With a generalized interpretation, we find that our proposed LE-EDF remains online optimal for scheduling MC job set, and is asymptotically optimal in its computational complexity (more efficient than the LP-based method). Even comparing to an optimal clairvoyant scheduler, we show that LE-EDF has a speedup of $3/4$, which is the best-known speedup result for MC job scheduling. For scheduling MC task sets,

existing scheduler named EDF-VD can be adapted regardless of the capability of self-monitoring or not. Some improvements can be made during the computing process, with better schedulability results shown experimentally via randomly generated sets.

In general, this dissertation extends the existing MC scheduling theory in several directions by proposing new schedulers, analyzing their properties thoroughly, and comparing to existing work. It has the potential to lead to more efficient design, analysis, and implementation of future real-time systems.

6.2 Other Contributions

In this section, some of the other major contributions made during my Ph.D. study will be highlighted. As they may not directly support our thesis, the introductions are kept in very light form — please refer to the publications for details.

6.2.1 A Comparison of MC Job Models

The Vestal model is widely used in the real-time scheduling community for representing mixed-criticality real-time workloads. When the total number of criticality levels exceed 2, Vestal model requires that multiple WCET estimates are obtained for each task.

Burns suggests (Burns, 2015) that being required to obtain too many WCET estimates may place an undue burden on system developers, and proposes a simplification of the Vestal model that makes do with just two WCET estimates per task. From a pragmatic perspective and in terms of ease of use, there are undoubted benefits in using the Burns model in preference to the Vestal model.

We reported on our attempts in (S. Baruah and Z. Guo, 2015) at comparing the two models – Vestals original model and Burns simplification – with regards to expressiveness, as well as schedulability and the tractability of determining schedulability. In our research, we are seeking to better understand whether the reduced expressiveness in Burn's model yields any analytical benefits in terms of reduced complexity of feasibility analysis, less schedulability loss, etc. Thus far, our

results have been negative we have not identified any such benefits when restricting our attention to MC instances that are characterized as collections of independent jobs.

6.2.2 Another Extension of the Vestal Model

The original Vestal model was proved very successful in identifying some of the core challenges that arise in resource-efficient scheduling of MC systems, and spawned a large body of research that proposed solutions to some of these challenges. However, this model has met with some criticism from systems engineers that it does not match their expectations in some important aspects.

In (Baruah et al., 2016), we focus upon one such aspect: in the event of some jobs executing beyond their LO-criticality WCET estimates, LO-criticality jobs should nevertheless be guaranteed some amount of execution prior to their deadlines. Followed by the initiative idea reported in (S. Baruah and A. Burns, 2014), we modified the specification and semantics of the Vestal model in two ways:

§1. While each task τ_i continues to be characterized by the two WCET parameters C_i^L and C_i^H, it is required that

1. If $\chi_i = \text{HI}$ then $C_i^H \geq C_i^L$ (this is as in the original Vestal model);
2. If $\chi_i = \text{LO}$, then $C_i^H \leq C_i^L$ (this is different).

§2. The run-time scheduling objectives are extended in the following manner to ensure a degraded (but non-zero) level of service for LO-criticality tasks in the event of HI-criticality tasks executing beyond their LO-crtiticality WCETs:

1. if each job of each task τ_i completes within C_i^L units of execution then all jobs complete by their deadlines; and
2. if a job of some HI-criticality task τ_i fails to complete despite being allowed to execute for C_i^L time units, then all jobs of all HI-criticality tasks τ_i should be allowed to execute for up to C_i^H units by their deadlines; additionally *all jobs of all* LO*-criticality tasks τ_i are guaranteed to receive at least C_i^H units of execution by their deadlines*.

Intuitively speaking, the WCET parameters of HI-criticality tasks are assumptions or *rely conditions* (Jones, 1981), and the WCET parameters of LO-criticality tasks are *corresponding guarantees* or budgets.

In (Baruah et al., 2016), we obtain a fluid model (see Sec. 3.3) based algorithm for the preemptive uniprocessor scheduling of dual-criticality task systems represented in this more general model, and prove that our algorithm has a speedup factor of $4/3$. Since this model is a generalization of the one for which the lower bound of $4/3$ on speedup was proved in (Baruah et al., 2012b, Theorem 5), it follows that no algorithm for scheduling the more general model may have a speedup bound smaller than $4/3$ and our algorithm is thus speedup-optimal. The MC task generator we used in this work is exactly the same as the one reported in Sec. 3.2.5.1, which has passed an Artifact Evaluation[1] process.

The contribution made in (Baruah et al., 2016) supports the central thesis in the sense that improvements could be made by refining existing models. It could be a good supplement for Chapter 3 — we choose to list it here since this work is done in parallel with the writing of this dissertation, and I am not the main contributor of (Baruah et al., 2016).

6.2.3 A CPS Case Study on EDF Schedulability of AVR tasks

Modern embedded systems broadly interact with physical environments. CPS are the intersection (not the union) of the physical and the cyber systems, where physical processes are often affected by computations and vice versa. CPS conjoins distinct disciplines, however, models that prevail in these distinct disciplines do not combine well (Lee, 2015). One of the most advanced and sophisticated models, the Adaptive Varying-Rate (AVR) task (Buttle, 2012), deals with the modeling of recurrent processes in CPS for which each activation of the recurrent process is triggered by the state of the physical system. Such processes abound in CPS: for example, height detection in avionic systems is activated more frequently at lower altitudes; sensor acquisition in mobile robots often depends on the robot location; and fuel injection in the Engine Control Unit (ECU) of an

[1]For additional details, please refer to http://ecrts.org/artifactevaluation.

automobile is dependent upon the position of each piston. We are among the first few researchers that started to investigate this model, and our ICCPS publication (Guo and Baruah, 2015b) is the first piece of work that thoroughly studies Earliest Deadline First (EDF) schedulability of AVR tasks.

A sufficient and fast schedulability test is shown for implicit systems, and its speedup factor (as a function of engine rotation speed) is derived. Under some practical assumptions, this result is further improved to be necessary and sufficient. For constrained systems (with relative deadlines smaller than periods), an attempt for demand based function analysis has been made by transforming into the digraph based task model. Schedulability experiments confirm that the proposed methods outperform the current state of the art from the perspective of schedulability ratio. Overall performance is further compared in these schedulability experiments with respect to changes in specific parameters, one at a time. Part of our theory results have been validated by a well-known research group in Europe via simulation (Biondi et al., 2015), and is being evaluated for adoption by the automotive industry (e.g., Volkswagen).

6.2.4 Solving MC Scheduling via a Neurodynamic Approach

Many novel recurrent neural network (RNN) models have recently been proposed for solving optimization problems with linear inequality constraints. These RNN models are often with very simple structures, and converge to the global optima rapidly. Due to the parallel nature structure of the RNNs, these models may be applied to parallel computing devices. Moreover, RNN based approaches have the potential of being implemented on hardware. As a result, the converging periods (into a stable state) of such systems can be extremely short comparing to the execution length of real-time jobs.

To investigate the potential of applying RNNs in real-time scheduling, in (Guo and Baruah, 2016), we apply one of the RNN models on a series of real-time job scheduling problems upon uniprocessors. We have presented rules for transformation and approximation from some typical NP-hard real-time scheduling problems into RNN solvable problems, and shown how they work out by

examples. Experimental studies suggest that the convergence time of the introduced neurodynamic system is likely to stay in a constant range when the size of job set grows, which indicates that our method may serve as the scheduler for large scale platforms, e.g., supercomputers, computing grids, and cloud centers. Based on the randomly generated 10000 job sets, comparison studies have been reported. It is evident that the proposed RNN based method outperforms EDF and Fixed Priority under overloaded conditions, while remains optimal (same as EDF) in non-overloaded conditions.

6.2.5 Other Publications During Ph.D. Study

I have been very fortunate to have had opportunities to work with truly outstanding researchers on various areas in the past 5 years, other than real-time systems (French et al., 2012) (Guo, 2015) (Guo, 2016), e.g., *Big Data* and *Bioinformatics* (Cheng et al., 2014) (Chen et al., 2014) (Cheng et al., 2013) (Liu et al., 2012a) (Crowley et al., 2015), *Neural Network* and *Computational Intelligence* (Guo et al., 2011b) (Guo et al., 2011a) (Liu et al., 2012b).

6.3 Future Directions

Mixed-criticality scheduling theory is so fundamental that it will remain attractive in the foreseeable future. Although this dissertation has answered several fundamental questions in mixed-criticality scheduling theory, many vast blanks in this area need to be filled. In this section, we try to list some limitations of our work, and point out some related and important future research directions.

More than two criticality levels. Although for some cases like MC job scheduling, we have provided nice schedulers for systems with an arbitrary number of criticality levels, most of our (and existing) work only apply to dual-criticality systems. In many cases, it is a huge step to improve from 2 to 3 — new techniques in both scheduling and analyzing may need to be introduced.

Restricted preemption. We only consider fully-preemptive systems in this dissertation. In some cases, in order to achieve the best theoretical result, we mimic a processor sharing scheme where the number of preemptions is potentially unlimited, which is impractical — each time a job gets

preempted and resumes execution, runtime overheads are incurred for managing scheduling queues and reloading cache lines. Too many preemptions often result in less predictable WCETs of tasks and more capacity waste due to the conservative assumptions made during the certification process. It is important to come up with schedulers with limited/restricted preemptions and studied via system level experiments.

Modeling the uncertainties. The pessimism during modeling process is unavoidable due to the uncertainty of system behaviors during run-time. We have tried to introduce probabilistic analysis into MC scheduling, yet with a lack of fundamental understanding of such uncertainties, which could be both epistemic (uncertainty in what we know, or do not know, about the system) and aleatory (uncertainty in the system itself). Probability theories cannot be directly applied to epistemic uncertainties — one potential way may be introducing uncertainties in our decisions (e.g., fuzzy theory, randomized cache), and is left as future work.

Dealing with heterogeneous platforms. Our varying-speed platform model does not easily extend to multiprocessor platforms. When some processors experiencing performance drop while some may not, we are actually facing a heterogeneous platform, and the schedulability upon such platforms is hard to achieve. We have reported some easy solution via existing techniques, while much remains to be done along this direction to make the results practical.

CPS based study. Advanced CPS will shape the interaction between human beings and the physical world — just as the world-wide-web shaped the interaction between human beings. However, CPS conjoins distinct disciplines, and models that prevail in these distinct disciplines do not combine well (Lee, 2015). As a result, new CPS models must be proposed as CPS evolves. It is our ability of understanding and analyzing the new model, as well as its fidelity (i.e., the degree to which the model imitates the system being modeled), that decides the value of the model. The models studied in this dissertation are quite general, which may not fit the need of designing specific CPS systems, and efforts could be spent on investigating more sophisticated models, e.g., the AVR task model (Buttle, 2012), the DAG based task model (Baruah et al., 2012c), etc.

BIBLIOGRAPHY

Abeni, L. and Buttazzo, G. (1999). Qos guarantee using probabilistic deadlines. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems (ECRTS'99)*.

Albers, K. and Slomka, F. (2004). An event stream driven approximation for the analysis of real-time systems. In *Proceedings of the EuroMicro Conference on Real-Time Systems (ECRTS'04)*.

Audsley, N. C. (1991). Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical report, The University of York, England.

Audsley, N. C. (1993). *Flexible Scheduling in Hard-Real-Time Systems*. PhD thesis, Department of Computer Science, University of York.

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

Bansal, N., Chan, H.-L., Khandekar, R., Pruhs, K., Schicber, B., and Stein, C. (2007). Non-preemptive min-sum scheduling with resource augmentation. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS '07)*.

Baruah, S. (2012). Certification-cognizant scheduling of tasks with pessimistic frequency specification. In *Proceedings of the 7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*.

Baruah, S. (2016). Mixed criticality schedulability analysis is highly intractable. Avaliable at `http://www.cs.unc.edu/~baruah/Submitted/02cxty.pdf`.

Baruah, S., Bonifaci, V., D'Angelo, G., Li, H., Marchetti-Spaccamela, A., Megow, N., and Stougie, L. (2010a). Scheduling real-time mixed-criticality jobs. In *Proceedings of the 35th International Symposium on the Mathematical Foundations of Computer Science (MFCS'10)*.

Baruah, S., Bonifaci, V., D'Angelo, G., Li, H., Marchetti-Spaccamela, A., Megow, N., and Stougie, L. (2012a). Scheduling real-time mixed-criticality jobs. *IEEE Transactions on Computers*, 61(8):1140–1152.

Baruah, S., Bonifaci, V., D'Angelo, G., Li, H., Marchetti-Spaccamela, A., van der Ster, S., and Stougie, L. (2012b). The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS'12)*.

Baruah, S., Bonifaci, V., D'Angelo, G., Marchetti-Spaccamela, A., van der Ster, S., and Stougie, L. (2011a). Mixed-criticality scheduling of sporadic task systems. In *Proceedings of the 19th Annual European Symposium on Algorithms (ESA'11)*.

Baruah, S., Bonifaci, V., Marchetti-Spaccamela, A., Stougie, L., and Wiese, A. (2012c). A generalized parallel task model for recurrent real-time processes. In *Proceedings of the 33rd IEEE Real-Time Systems Symposium (RTSS'12)*.

Baruah, S. and Burns, A. (2006). Sustainable scheduling analysis. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS'06)*.

Baruah, S., Burns, A., and Davis, R. (2011b). Response-time analysis for mixed criticality systems. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS'11)*.

Baruah, S., Burns, A., and Guo, Z. (2016). Scheduling mixed-criticality systems to guarantee some service under all non-erroneous behaviors. In *Proceedings of the 28th Euromicro Conference on Real-Time Systems (ECRTS'16)*.

Baruah, S. and Chattopadhyay, B. (2013). Response-time analysis of mixed criticality systems with pessimistic frequency specification. In *Proceedings of the 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'13)*.

Baruah, S., Cohen, N., Plaxton, C., and Varvel, D. (1996). Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625.

Baruah, S., Easwaran, A., and Guo, Z. (2015). MC-Fluid: simplified and optimally quantified. In *Proceedings of the 36th IEEE Real-Time Systems Symposium (RTSS'15)*.

Baruah, S. and Guo, Z. (2013). Mixed-criticality scheduling upon varying-speed processors. In *Proceedings of the 34th IEEE Real-Time Systems Symposium (RTSS'13)*.

Baruah, S. and Guo, Z. (2014). Scheduling mixed-criticality implicit-deadline sporadic task systems upon a varying-speed processor. In *Proceedings of the 35th IEEE Real-Time Systems Symposium (RTSS'14)*.

Baruah, S., Li, H., and Stougie, L. (2010b). Towards the design of certifiable mixed-criticality systems. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'10)*.

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

Bernat, G., Colin, A., and Petters, S. (2003). PWCET: a tool for probabilistic worst-case execution time analysis of real-time systems. Report - University of York Department of Computer Science YCS.

Bini, E. and Buttazzo, G. (2005). Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154.

Biondi, A., Buttazzo, G., and Simoncelli, S. (2015). Feasibility analysis of engine control tasks under EDF scheduling. In *Proceedings of the 27th Euromicro Conference on Real-Time Systems (ECRTS'15)*.

Bull, D., Das, S., Shivshankarand, K., Dasika, G., Flautner, K., and Blaauw, D. (2010). A power-efficient 32b ARM ISA processor using timing-error detection and correction for transient-error tolerance and adaptation to PVT variation. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC'10)*.

Burns, A. (2015). An augmented model for mixed criticality. *S. K. Baruah, L. Cucu-Grosjean, R. I. Davis, and C. Maiza, editors, Mixed Criticality on Multicore/Manycore Platforms (Dagstuhl Seminar 15121)*, 5. Schloss Dagstuhl Leibniz-Zentrum fuer Informatik.

Burns, A. and Davis, R. (2013). Mixed criticality on controller area network. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS'13)*.

Burns, A. and Davis, R. (2016). Mixed-criticality systems: A review. Avaliable at `http://www-users.cs.york.ac.uk/~burns/review.pdf`.

Buttazzo, G. (2005). *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer, second edition.

Buttazzo, G., Bini, E., and Buttle, D. (2014). Rate-adaptive tasks: Model, analysis, and design issues. In *Proceedings of the International Conference on Design, Automation and Test in Europe (DATE'14)*.

Buttle, D. (2012). Real-time in the prime-time. In *Keynote speech given at the 24th Euromicro Conference on Real-Time Systems (ECRTS'12)*.

Cazorla, F., Quinones, E., Vardanega, T., Cucu-Grosjean, L., Triquet, B., Bernat, G., Berger, E., Abella, J., Wartel, F., Houston, M., Santinelli, L., Kosmidis, L., Lo, C., and Maxim, D. (2013). Proartis: Probabilistically analysable real-time systems. *ACM Trans. Embedded Computing Systems (TECS)*, 12(2):1–26.

Chen, W., Zhang, X., Guo, Z., Shi, Y., and Wang, W. (2014). Graph regularized dual lasso for robust eqtl mapping. *Bioinformatics*, 30(12):i139–i148.

Cheng, W., Zhang, X., Guo, Z., Shi, Y., and Wang, W. (2014). Graph regularized dual lasso for robust eqtl mapping. In *Proceedings of the 22nd Annual International Conference on Intelligent Systems for Molecular Biology (ISMB'14)*.

Cheng, W., Zhang, X., Guo, Z., Wu, Y., Sullivan, P., and Wang, W. (2013). Flexible and robust co-regularized multi-domain graph clustering. In *Proceedings of the 19th ACM Conference on Knowledge Discovery and Data Mining (SIGKDD'13)*.

Cho, H., Ravindran, B., and Jensen, E. D. (2006). An optimal real-time scheduling algorithm for multiprocessors. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS'06)*.

Crowley, J., Zhabotynsky, V., Sun, W., Huang, S., Pakatci, I. K., Kim, Y., Wang, J. R., Morgan, A. P., Calaway, J. D., Aylor, D. L., Yun, Z., Bell, T. A., Buus, R. J., Calaway, M. E., Didion, J. P., Gooch, T. J., Hansen, S. D., Robinson, N. N., Shaw, G. D., Spence, J. S., Quackenbush, C. R., Barrick, C. J., Nonneman, R. J., Kim, K., Xenakis, J., Xie, Y., Valdar, W., Lenarcic, A. B., Wang, W., Welsh, C. E., Fu, C.-P., Zhang, Z., Holt, J., Guo, Z., Threadgill, D. W., Tarantino, L. M., Miller, D. R., Zou, F., McMillan, L., Sullivan, P. F., and de Villena, F. P.-M. (2015). Analyses of allele-specific gene expression in highly divergent mouse crosses identifies pervasive allelic imbalance. *Nature Genetics*, 47:353–360.

Cucu-Grosjean, L. (2013). Independence - a misunderstood property of and for probabilistic real-time systems. In *N. Audsley and S. Baruah, editors, Real-Time Systems: the past, the present and the future*.

Cucu-Grosjean, L., Santinelli, L., Houston, M., Lo, C., Vardanega, T., Kosmidis, L., Abella, J., Mezzetti, E., Quinones, E., and Cazorla, F. (2012). Measurement-based probabilistic timing analysis for multi-path programs. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS'12)*.

David, L. and Puaut, I. (2004). Static determination of probabilistic execution times. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS'04)*.

Davis, R. I., Santinelli, L., Altmeyer, S., Maiza, C., and Cucu-Grosjean, L. (2013). Analysis of probabilistic cache related pre-emption delays. *Proceedings of the 25th IEEE Euromicro Conference on Real-Time Systems (ECRTS'13)*.

Dertouzos, M. (1974). Control robotics: the procedural control of physical processors. In *Proceedings of the IFIP Congress*.

Díaz, J., Garcia, D., Lee, C., Bello, L., López, J., and Mirabella, O. (2002). Stochastic analysis of periodic real-time systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*.

Edgar, S. and Burns, A. (2001). Statistical analysis of wcet for scheduling. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*.

F. Eisenbrand and T. Rothvoß (2010). Edf-schedulability of synchronous periodic task systems is conp-hard. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SDA'10)*.

French, A., Guo, Z., and Baruah, S. (2012). Scheduling mixed-criticality workloads upon unreliable processors. In *Proceedings of the 11th Workshop on Models and Algorithms for Planning and Scheduling Problems (MAPSP'12)*.

Funk, S., Levin, G., Sadowski, C., Pye, I., and Brandt, S. (2011). DP-Fair: a unifying theory for optimal hard real-time multiprocessor scheduling. *Real-Time Systems*, 47(5):389–429.

Gardner, M. and Liu, J. (1999). Analyzing stochastic fixed-priority real-time systems. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*.

Garey, M. and Johnson, D. (1979). *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman and company, NY.

Griffin, D. and Burns, A. (2010). Realism in statistical analysis of worst case execution times. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET'10)*.

Guan, N., Ekberg, P., Stigge, M., and Yi, W. (2011). Effective and efficient scheduling for certifiable mixed criticality sporadic task systems. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS'11)*.

Guan, N., Ekberg, P., Stigge, M., and Yi, W. (2013). Improving the scheduling of certifiable mixed-criticality sporadic task systems. Technical Report 2013-008, Department of Information Technology, Uppsala University.

Guo, Y., Sun, F., and Guo, Z. (2011a). Control allocation of flying-wing with multi-effectors based on ts fuzzy model. In *Proceeding of the 3rd International Conference on Mechanical and Electrical Technology (ICMET'11)*.

Guo, Z. (2015). MC scheduling on varying-speed processors. In *Dagstuhl Seminar 15121: Mixed Criticality on Multicore/Manycore Platforms*.

Guo, Z. (2016). Mixed-criticality scheduling on varying-speed platforms with bounded performance dropping rate. In *Proceedings of the 2nd IEEE International Conference on Smart Computing (SMARTCOMP'16) (WiP Session)*.

Guo, Z. and Baruah, S. (2013). Mixed-criticality scheduling upon unmonitored unreliable processors. In *Proceedings of the 8th IEEE International Symposium on Industrial Embedded Systems (SIES'13)*.

Guo, Z. and Baruah, S. (2014a). Mixed-criticality scheduling upon varying-speed multiprocessors. *Leibniz Transactions on Embedded Systems*, 1(2):3:1–3:19.

Guo, Z. and Baruah, S. (2014b). Mixed-criticality scheduling upon varying-speed multiprocessors. In *Proceedings of the 12th IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC'14)*.

Guo, Z. and Baruah, S. (2015a). The concurrent consideration of uncertainty in WCETs and processor speeds in mixed-criticality systems. In *the 23rd International Conference on Real-Time and Network Systems (RTNS'15)*.

Guo, Z. and Baruah, S. (2015b). Uniprocessor EDF scheduling of avr task systems. In *Proceedings of the 6th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS'15)*.

Guo, Z. and Baruah, S. (2016). A neurodynamic approach for real-time scheduling via maximizing piecewise linear utility. *IEEE Trans. Neural Netw. and Learn. Sys.*, 27(2):238–248.

Guo, Z., Liu, Q., and Wang, J. (2011b). A one-layer recurrent neural network for pseudoconvex optimization subject to linear equality constraints. *IEEE Trans. Neural Netw.*, 22:1892–1900.

Guo, Z., Santinalli, L., and Yang, K. (2015). EDF schedulability analysis on mixed-criticality systems with permitted failure probability. In *Proceedings of the 21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'15)*.

Hansen, J., Hissam, S., and Moreno, G. (2009). Statistical-based wcet estimation and validation. In *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*.

Hansen, J., Lehoczky, J., Zhu, H., and Rajkumar, R. (2002). Quantized EDF scheduling in a stochastic enviroment. In *Proceedings of the 16th IEEE International Parallel and Distributed Processing Symposium (IPDPS'02)*.

Hardy, D. and Puaut, I. (2013). Static probabilistic worst case execution time estimation for architectures with faulty instruction caches. In *Proceedings of the 21st International Conference on Real-Time and Networked Systems (RTNS'13)*.

Holman, P. and Anderson, J. H. (2005). Adapting pfair scheduling for symmetric multiprocessors. *J. Embedded Comput.*, 1(4):543–564.

Horvath, E., Lam, S., and Sethi, R. (1977). A level algorithm for preemptive scheduling. *Journal of the ACM*, 24(1):32–43.

Jones, C. (1981). *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University. Printed as Programming Research Group, Technical Monograph 25.

Kalyanasundaram, B. and Pruhs, K. (2000). Speed is as powerful as clairvoyance. *Journal of the ACM*, 37(4):617–643.

Karmakar, N. (1984). A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395.

Kellerer, H., Pferschy, U., and Pisinger, D. (2004). *Knapsack Problems*. Springer.

Khachiyan, L. (1979). A polynomial algorithm in linear programming. *Dokklady Akademiia Nauk SSSR*, 244:1093–1096.

Lawler, E. L. (1973). Optimal sequencing of a single machine subject to precedence constraints. *Management Science*, 19(5):544–546.

Lee, E. A. (2015). The past, present and future of cyber-physical systems: A focus on models. *Sensors*, 15(3):4837–4869.

Lee, J., Phan, K.-M., Gu, X., Lee, J., Easwaran, A., Shin, I., and Lee, I. (2014). MCFluid: Fluid model-based mixed-criticality scheduling on multiprocessors. In *Proceedings of the 35th IEEE Real-Time Systems Symposium (RTSS'14)*.

Lehoczky, J. (1996). Real-time queueing theory. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS'96)*.

Lenstra, J., A.H.G. Rinnooy Kan, and Brucker, P. (1977). Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362.

Leung, J. (2004). *Handbook of scheduling: algorithms, models, and performance analysis*. CRC Press.

Li, H. and Baruah, S. (2010). An algorithm for scheduling certifiable mixed-criticality sporadic task systems. In *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS'10)*.

Li, H. and Baruah, S. (2012). Global mixed-criticality scheduling on multiprocessors. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS'12)*.

Liu, C. and Layland, J. (1973). Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61.

Liu, E., Guo, Z., Zhang, X., Jojic, V., and Wang, W. (2012a). Metric learning from relative comparisons by minimizing squared residual. In *Proceedings of the 12th IEEE International Conference on Data Mining (ICDM'12)*.

Liu, J. (2000). *Real-Time Systems*. Prentice-Hall, Inc., Upper Saddle River, New Jersey 07458.

Liu, Q., Guo, Z., and Wang, J. (2012b). A one-layer recurrent neural network for constrained pseudoconvex optimization and its application for dynamic portfolio optimization. *Neural Networks*, 26:99–109.

Lu, Y., Nolte, T., Bate, I., and Cucu-Grosjean, L. (2012). A statistical response-time analysis of real-time embedded systems. In *Proceedings of the 33rd IEEE Real-Time Systems Symposium (RTSS'12)*.

Manolache, S., Eles, P., and Peng, Z. (2004). Schedulability analysis of applications with stochastic task execution times. *ACM Trans. Embedded Computing Systems (TECS)*, 3(4):706–735.

Maxim, D., Buffet, O., Santinelli, L., Cucu-Grosjean, L., and Davis, R. (2011). Optimal priority assignment algorithms for probabilistic real-time systems. In *Proceedings of the 19th International Conference on Real-Time and Networked Systems (RTNS'11)*.

Maxim, D. and Cucu-Grosjean, L. (2013). Response time analysis for fixed-priority tasks with multiple probabilistic parameters. In *Proceedings of the 34th IEEE Real-Time Systems Symposium (RTSS'13)*.

Maxim, D., Houston, M., Santinelli, L., Bernat, G., Davis, R. I., and Cucu-Grosjean, L. (2012). Re-sampling for statistical timing analysis of real-time systems. In *the 20th International Conference on Real-Time and Network Systems (RTNS'12)*.

Melani, A., Noulard, E., and Santinelli, L. (2013). Learning from probabilities: Dependences within real-time systems. In *the 8th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'13)*.

Mok, A. K. (1983). *Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment*. PhD thesis, Massachusetts Institute of Technology.

Mok, A. K. (1988). Task management techniques for enforcing ED scheduling on a periodic task set. In *Proceedings of the 5th IEEE Workshop on Real-Time Software and Operating Systems*.

Niz, D., Lakshmanan, K., and Rajkumar, R. (2009). On the scheduling of mixed-criticality real-time task sets. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, RTSS 2009.

S. Baruah and A. Burns (2014). Towards a more practical model for mixed criticality systems. In *Proceedings of the Workshop on Mixed-Criticality Systems (WMC'14)*.

S. Baruah and Z. Guo (2015). Mixed-criticality job models: a comparison. In *Proceedings of the Workshop on Mixed-Criticality Systems (WMC'15)*.

Santinelli, L., Buttazzo, G., and Bini, E. (2011). Multi-moded resource reservation. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.

Santinelli, L., Morio, J., Dufour, G., and Jacquemart, D. (2014). On the sustainability of the extreme value theory for WCET estimation. In *the 14th International Workshop on Worst-Case Execution Time Analysis (WCET'14)*.

Slijepcevic, M., Kosmidis, L., Abella, J., Nones, E. Q., and Cazorla, F. J. (2013). Dtm: Degraded test mode for fault-aware probabilistic timing analysis. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS'13)*.

Socci, D., Poplavko, P., Bensalem, S., and Bozga, M. (2013). Mixed critical earliest deadline first. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems (ECRTS'13)*.

Souyris, J., Pavec, E., Himbert, G., Jegu, V., and Borios, G. (2005). Computing the worst case execution time of an avionics program by abstract interpretation. In *Proceedings of the 5th International Workshop on Worst-Case Execution Time Analysis (WCET'05)*.

Stajano, F. and Anderson, R. (2000). The grenade timer: fortifying the watchdog timer against malicious mobile code. In *Proceedings of 7th International Workshop on Mobile Multimedia Communications (MoMuC'00)*.

Stoimenov, N., Thiele, L., Santinelli, L., and Buttazzo, G. (2010). Resource adaptations with servers for hard real-time systems. In *Proceedings of 10th International Conference On Embedded Software (EMSOFT'10)*.

Tia, T., Deng, Z., Storch, M., Sun, J., Wu, L., and Liu, J. (1995). Probabilistic performance guarantee for real-time tasks with varying computation times. In *Proceedings of 1st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'95)*.

Vestal, S. (2007). Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the 28th IEEE Real-Time Systems Symposium (RTSS'07)*.

Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., and Stenström, P. (2008). The worst-case execution-time problem – overview of methods and survey of tools. *ACM Trans. Embedded Computing Systems (TECS)*, 7(3):1–53.

Zhu, H., Hansen, J., Lehoczky, J., and Rajkumar, R. (2002). Optimal partitioning for quantized EDF scheduling. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*.