DISS. ETH NO. 24023

Design and Optimization of Mixed-Criticality Systems

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by
PENGCHENG HUANG
Master of Science,
Delft University of Technology

born on May 26, 1987
citizen of China

accepted on the recommendation of
Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Alan Burns, co-examiner

2016

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

TIK-SCHRIFTENREIHE NR. 168

Pengcheng Huang

Design and Optimization of Mixed-Criticality Systems

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A dissertation submitted to
ETH Zurich
for the degree of Doctor of Sciences

Diss. ETH No. 24023

Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Alan Burns, co-examiner

Examination date: December 13, 2016

致我的家人——一切愛皆始於家庭。

Abstract

Mixed-Criticality is emerging as a significant trend for safety-critical systems, especially in automotive and avionics industries. Conventionally, those systems are designed as multiple sub-systems of *distinct* criticality or importance levels. With the ever-increasing demand of system functionalities and the shift of the semiconductor industry to more powerful (multi-core) platforms, consolidating/mixing functionalities of different criticality levels in a common hosting platform is appealing – system costs induced by size, weight and power consumptions could be potentially greatly reduced in the mixed-criticality setting.

The benefits brought by mixed-criticality systems are, however, accompanied by a multitude of new challenges. Most noticeably, due to resource sharing, functionalities of different criticality levels can interfere with each other, jeopardizing their guarantees made in isolation. To address this, new mixed-criticality models/protocols and corresponding scheduling techniques need to be developed to provide adequate isolation among criticality levels, and their limits (as well as potential extensions) need to be understood. Many traditional design issues for conventional embedded and/or safety-critical systems need to be rethought in the mixed-criticality era. For example, in addition to common timing threats considered for time-critical systems, hardware/software faults also need to be considered in the design of real-time mixed-safety-critical systems. In order to increase the system power efficiency, it is necessary to extend conventional energy minimization techniques to emerging mixed-criticality applications while considering the peculiarities of such systems.

As a step towards solving the above challenges, this thesis presents a whole stack of solutions to model, design and optimize mixed-criticality systems, in areas regarding real-time, fault-tolerance and energy-efficiency. Specifically, we make three main contributions:

1 We design the first mixed-criticality models to improve the service guarantee for less critical tasks in urgent scenarios – existing solutions commonly assume to drop all those tasks when any critical task overruns, which could be hardly acceptable in practice. In particular, we propose service adaptation, detailed modeling through interference constraint graphs, and processor over-clocking to adaptively degrade the system service in urgent scenarios. We show how common scheduling techniques like fixed-priority (FP)

and earliest deadline first (EDF) can be extended under those models and demonstrate considerable performance improvements compared to existing solutions. To further understand the limit of existing mixed-criticality models in line with industrial practices (i.e., with static temporal isolation among criticality levels), we present optimal scheduling techniques and we theoretically quantify the schedulability loss of those models.

2 We present the first mixed-criticality framework, where fault-tolerance, real-time requirements and runtime adaptation are jointly considered to achieve a safe system design. Assuming hardware/software faults, we adopt task re-execution (single-core) and replication (multi-core) as our fault-tolerance techniques and explicitly follow safety standards to model system safety requirements on different criticality levels. To further deal with urgent scenarios where critical tasks do not succeed after a certain number of trials, we propose runtime adaptations to reallocate system resources to critical tasks in such scenarios. Based on this, we present fault-tolerant mixed-criticality scheduling techniques and corresponding analysis techniques to meet both, safety and real-time requirements. Our solutions work on single-core and multi-core platforms, demonstrating the advantages of runtime adaptations and revealing important findings on the impact of commonly assumed mixed-criticality reconfigurations on the system feasibility.

3 We develop the first dynamic voltage and frequency scaling (DVFS) techniques to improve energy efficiency for mixed-criticality systems. We show that fundamental trade-offs exist for this problem: DVFS can help the system to speedup in order to overcome the urgent scenarios where critical tasks overrun, which further allows the system to relax (slow down) and save dynamic energy in nominal scenarios where tasks do not overrun. Assuming EDF based scheduling, we present optimal and heuristic solutions in a general setting firstly on a single-core, considering both leakage and dynamic energy consumptions and all different system operation scenarios. We then develop energy-aware mixed-criticality task mapping techniques to extend our single-core solutions to multi-core platforms. Our solutions demonstrate considerable energy savings for both synthetic task sets and a realistic industrial use-case, while revealing a rather surprising finding – the industrial best practice of spatially isolating different criticality levels almost has comparable energy savings to mixing them on each core.

Zusammenfassung

Mixed-Criticality ist ein bedeutender Trend im Bereich sicherheitskritischer Systeme, allem voran in der Automobil- und Avionikindustrie. Konventionelle Systeme bestehen aus mehreren Teilsystemen, welche *unterschiedliche* Kritikalitäts-/Wichtigkeitsniveaus abdecken. Mit steigender Nachfrage nach erweiterter Systemfunktionalität und der Verlagerung der Prozessorindustrie auf leistungsfähigere (Multi-Core) Plattformen ist es vielversprechend die Funktionalitäten unterschiedlicher Kritikalitätsniveaus auf einer gemeinsamen Plattform zu konsolidieren. Durch Grösse, Gewicht und Energieverbrauch bedingte Systemkosten lassen sich in einem Mixed-Criticality Setting möglicherweise stark reduzieren.

Die Vorteile, welche Mixed-Criticality Systeme bieten, werden jedoch von einer Vielzahl neuer Herausforderungen begleitet. Besonders durch Ressourcen-Sharing können sich verschiedene Kritikalitätsniveaus gegenseitig stören, obwohl durch Isolation diese gegenseitige Beeinflussung ausgeschlossen werden soll. Um diese Probleme zu adressieren, müssen neue Mixed-Criticality Modelle und Protokolle, sowie entsprechende Scheduling-Techniken entwickelt und deren Grenzen (und mögliche Erweiterungen) verstanden werden um genügende Isolation zwischen den unterschiedlichen Kritikalitätsniveaus zu garantieren. Viele typischen Designfragen konventioneller eingebetteter und/oder sicherheitskritischer Systeme müssen in der Mixed-Criticality Ära überdacht werden. So müssen zum Beispiel zusätzlich zu den üblichen Tasküberlauf in zeitkritischen Systemen auch Hardware- und Software-Fehler in der Designphase von Echtzeit Mixed-Criticality Systemen bedacht werden. Um ebenfalls die Energieeffizienz solcher Systeme zu verbessern, ist es ebenso notwendig konventionelle Energiesparmechanismen für zukünftige Mixed-Criticality Systeme zu erweitern.

Diese Disseration stellt einen ersten Schritt zur Lösung dieser Probleme dar. Dabei werden eine Reihe von Lösungen zur Modellierung, dem Design und der Optimierung von Mixed-Criticality Systemen in den Bereichen Echtzeitverarbeitung, Fehlertoleranz und Energieeffizienz präsentiert. Im Einzelnen werden folgende Beiträge gemacht:

1 Wir entwerfen die ersten Mixed-Criticality Modelle welche die Servicegarantie weniger kritischer Tasks in dringenden Situationen verbessern. Im Gegensatz dazu gehen bestehende Lösungen davon aus, dass weniger kritische Tasks automatisch abgebrochen werden sobald

ein kritischer Task überläuft, was in der Praxis kaum umsetzbar ist. Insbesondere schlagen wir dynamische Serviceadaptierung, detaillierterer Modellierung mittels Interferenz Bedingungsgraphen und Prozessorübertaktung vor, um den Systemservice in dringenden Situationen zu adaptieren. Wir zeigen auf wie bekannte Schedulingalgorithmen wie Fixed-Priority (FP) und Earliest-Deadline-First (EDF) mithilfe dieser Modelle erweitert werden können und demonstrieren dass diese die Performance gegenüber bestehenden Lösungen erheblich verbessern. Ebenso präsentieren wir optimale Schedulingtechniken und quantifizieren die Einbussen auf die Schedulability um die Einschränkungen bestehender Mixed-Criticality Modellen im Zusammenhang mit der Industriepraxis, wie zum Beispiel zeitliche Isolation der Kritikaliätsniveaus, zu verstehen.

2 Wir präsentieren das erste Mixed-Criticality Framework welches Echtzeit-Anforderungen und Serviceanpassung zur Laufzeit gemeinsam betrachtet um ein sicheres Systemdesign zu erzielen. Unter der Annahme von Hardware- und Softwarefehlern Übernehmen wir die Techniken Task-Wiederausführung (Single-Core) und Replikation (Multi-Core) um Fehlertoleranz zu garantieren und folgen den strikten Sicherheitsstandards um die Sicherheitsanforderungen der verschiedenen Kritikalitätsniveaus zu modellieren. Weiter behandeln wir dringende Situationen in welchen kritische Tasks auch nach einigen Wiederausführungen nicht erfolgreich abgearbeitet wurden. In diesen Fällen schlagen wir vor dynamisch weitere System-Ressourcen für diese kritischen Tasks zu allozieren. Darauf aufbauend präsentieren wir fehlertolerante Mixed-Cricicality Schedulingalgorithmen and entsprechende Analysetechniken um Sicherheits- und Echtzeitanforderungen zu erfüllen. Die gezeigte Lösung arbeitet sowohl auf Single-Core als auch Multi-Core Systemen und liefert wichtige Erkenntnisse über den Einfluss typischer Rekonfigurationsmassnahmen auf die Umsetzbarkeit von Mixed-Criticality Systemen.

3 Wir entwickeln die erste dynamische Spannungs- und Frequenzskalierung (Dynamic Voltage and Frequency Scaling, DVFS) Technik für Mixed-Criticality Systeme und zeigen auf, welche grundlegenden Kompromisse für dieses Problem getroffen werden müssen: DVFS kann helfen die Ausführung zu beschleunigen um dringende Situationen zu überwinden und hilft zudem das System zu verlangsamen um Energie einzusparen wenn kritische Tasks keine erhöhten Ausführungszeiten haben. Wir präsentieren eine optimale und heuristische Lösung für EDF basiertes Scheduling für Single-Core Architekturen unter Einbeziehung von Leakage und dynamischem Energieverbrauch und

allen möglichen Systemzuständen. Wir entwicklen energiebewusste Mixed-Criticality Task-Mapping Algorithmen zur Erweiterung der Single-Core Lösung auf Multi-Core Plattformen. Die Evaluation unserer Lösung zeigt signifikante Energieeinsparungen sowohl für synthetische Task-Sets, als auch realistische industrielle Anwendungen. Interessanterweise zeigt die Anwendung der gleichen Massnahmen auf die industrielle Best Practice räumlicher Isolation annähernd die gleichen Energieeinsparungen.

Acknowledgments

I owe my growth as a researcher to Prof. Lothar Thiele, which would have been limited without the opportunity he offered and his support during my PhD. He set up a good example for me, not only as a top researcher, but also as an educator and a colleague. Looking back, I have benefited a lot from the working experiences with him. For this, I am highly indebted.

I would like to express my sincere thanks to Prof. Alan Burns for helping to review this thesis and for traveling to Zürich in the midst of pressing affairs to serve on the committee board.

I feel very lucky to have many colleagues/friends supporting me both at work and in private lives. I would like to thank Georgia Giannopoulou, for sharing the office room with me in the past years, for her encouragement at the beginning of my PhD and her great kindness offered on an almost daily basis. I would also like to thank Pratyush Kumar, Hoeseok Yang, Lukas Sigrist, Andres Gomez, Stefan Draskovic, Rehan Ahmed, Balz Maag, Romain Jacob, Zimu Zhou, and other current and former TEC members – for a lot of discussions we had on both research and random topics – they were really rewarding experiences.

Beyond collaborators on the papers in this thesis, I would like to thank Lukas Sigrist for translating the thesis abstract into German, Andres Gomez for his suggestions on English writing and Luyuan Zeng for conducting the experiments in Chapter 3.

I would also take this chance to thank Beat Futterknecht, Federico Ferrari and Jan Beutel – without all their help, settling down in Zurich would have been much harder. My thanks extend to Brütsch Friederike for organizing the travel of my co-examiner.

Last but not least, I would like to express my gratitude to my family – I am indebted and grateful to my grandparents Songming and Guifang, my parents Zhi and Xinfang, and my sister Shengnan. My most special thanks go to Herma, Shan and Jing – for offering me a place called home.

The work presented in this thesis was supported in part by the EU FP7 project CERTAINTY. This support is gratefully acknowledged.

Contents

List of Figures

List of Tables

1

Introduction

Computer systems are evolving into complex mixed-criticality systems [BD16], in a natural process to enrich system functionalities by integrating subsystems/tasks/functionalities of differing importance/criticality levels. In the embedded system industry, such a trend is further intensified by the fact that embedded processors are nowadays shifting aggressively towards powerful, general purpose (multi-core) platforms [BDM09]. This creates a great platform base to enable mixed-criticality consolidation. Consequently, in the past few years, a lot of effort has been spent in both academia and industry (especially automotive and avionics [L+12]) on designing those emerging mixed-criticality systems. At the heart of their addressed problems are the following questions: How are mixed-criticality systems fundamentally different from conventional systems? How can existing design methods/methodologies (regarding system modeling, reliability, energy optimization, etc.) be extended in designing those new systems?

Implications of Mixed-Criticality: An answer to the above questions is certainly related to the definition of "criticality". Naturally, there exist multiple interpretations – criticality for a laptop functionality would merely mean user preference, while criticality for an aircraft flight control system directly relates to system safety (i.e., high costs of failures, being either social or economical). It is not difficult to see that the problem would be much harder in the latter case, as the design constraints are much more stringent. We will hence focus on mixed-safety-critical systems, the developments of which are currently strongly pushed in automotive and avionics industries [L+12].

Subsequently, one needs to understand the implications of "mixing"

functionalities of varying criticality on the same platform. The foremost problem would be that those functionalities can now interfere with each other on shared resources, jeopardizing their guarantees made in isolation – a strong argument that the new approach is in contradiction with the common wisdom that, in a mixed-safety-critical setting, the best option is to segregate functionalities of different importances [EN16]. In fact, it is still possible to guarantee system safety under "mixing", at the cost of certifying all functionalities to the highest criticality level. Unfortunately, this is often economically infeasible.

Potential Solutions: One possible solution is to achieve an equivalence of isolation by monitoring and regulating task behaviors in a system, as similarly done for traditional fault-tolerant systems [DGR04]. This way, we can for example ensure that tasks do not exceed their measured worst-case execution times at runtime, i.e., they do not interfere with each other's real-time guarantees made offline. To further take into account that tasks are of different importance levels, their parameters should be estimated in a way that reflects their criticality levels, e.g., a higher criticality indicates a more pessimistic time measurement.

Another solution is to simply enforce conventional isolation methods [AB98] among different criticality levels, e.g., by server-based scheduling techniques [SEL08]. However, since a multitude of resources are shared across applications on a modern computing platform (e.g., processing cores, memory systems, buses etc.), this would imply an exhaustive isolation on all shared resources, giving rise to poor resource utilization. Therefore, new isolation methods need to be developed to allow criticality isolation in a resource efficient manner.

A Wider Perspective: We are addressing the design of systems, with the characteristic of being mixed-critical. Thus, many conventional design issues such as real-time performance, system reliability, and energy conservation need to be addressed in the mixed-criticality context. A common misconception, however, is that resource efficiency is only a secondary design goal for safety-critical systems. This is hardly true from a business point of view. As an example, the automotive industry is highly competitive and cost sensitive [Sud01], and it is of ultimate importance to design safe mixed-criticality systems while simultaneously reducing system costs. In this context, established techniques/methodologies [Mar10] need to be extended while taking into account the peculiarities of mixed-criticality systems.

Thesis Contribution and Organization: This thesis provides a whole stack of technologies to specify and schedule mixed-criticality systems with a focus on real-time, fault-tolerance and energy efficiency. Specifically, three main contributions are made throughout this thesis:

- Chapter 2 proposes the first mixed-criticality models, as well as corresponding scheduling techniques, in order to improve the service guarantee for less critical tasks in urgent scenarios. Existing solutions commonly assume to drop all those tasks when any critical task overruns – a solution that improves system resource efficiency, but is hardly acceptable in practice. For the proposed service adaptation, the definition and use of interference constraint graphs and processor over-clocking, we demonstrate both theoretical and experimental improvements over existing methods. To further understand the limit of existing mixed-criticality models in line with industrial practices (i.e., with static temporal isolation among criticality levels), we present optimal scheduling techniques and theoretically quantify the schedulability loss of those models.
- Chapter 3 presents the first mixed-criticality framework, unifying considerations of fault-tolerance, real-time and runtime adaptation to achieve a safe and efficient system design. Under hardware/software faults, we adopt task re-execution and replication to achieve fault-tolerance and explicitly follow safety standards to model system safety requirements on different criticality levels. We further adopt runtime adaptations to reallocate system resources to critical tasks, when they do not succeed after a certain number of trials. We then present fault-tolerant mixed-criticality scheduling techniques and corresponding analyses to satisfy both safety and task deadlines. Our proposed solutions work on single-cores and multi-cores, demonstrating the advantages of runtime adaptations and revealing important findings on the impact of commonly assumed mixed-criticality reconfigurations on system feasibility.
- Chapter 4 develops the first energy conservation techniques for mixed-criticality systems, leveraging dynamic voltage and frequency scaling (DVFS) capabilities of modern computing platforms. We show that fundamental trade-offs exist for this problem: DVFS helps the system to speedup to overcome urgent scenarios where critical tasks overrun; this further allows the system to relax (slow down) and save energy when tasks do not overrun. We provide heuristic and optimal solutions to minimize system energy on a single-core under EDF scheduling, as well as the extension to multi-core with energy-aware task mapping. Our solutions demonstrate considerable energy savings for both, synthetic and industrial systems, while revealing an interesting finding – the industrial practice of spatially isolating criticality levels almost has comparable energy savings to mixing them on each core.

Chapter 5 summarizes this thesis and outlines future directions.

2

Mixed-Criticality Models – Limits, Extensions and Scheduling

Mixed-criticality systems advocate to use a common computing platform to host applications of different criticality or importance levels (we shall refer to all tasks with one criticality level as one task class, whenever it is convenient). Such an approach, despite its great potential in reducing system costs (size, weight and power), poses significant new challenges on the system design – applications can interfere with each other on shared resources, jeopardizing their guarantees made in isolation. Addressing this would require a whole stack of solutions ranging from high level modeling and specification to low level deployment. This chapter is dedicated to understanding and solving the modeling of mixed-criticality systems and shall focus on system real-time properties. The core of such modeling shall be straightforwardly extensible when other system properties are considered, see e.g., Chapter 3 for reliability.

The central problem here is to provide a high level specification of mixed-criticality systems, clarifying most importantly the following issues: (1) What guarantee should we provide for each criticality level? (2) Under which conditions should the respective guarantees on various criticality levels be made? (3) With such a model once given, it is further important to understand how to schedule the system accordingly and when can we guarantee task deadlines. For the first question, an intuitive general answer is already given by common safety standards [do11] – the rigorousness of guarantees increases as criticality levels ascend. This means that, from a real-time perspective, more pessimistic worst-case execution time (WCET) measurements should be adopted on a higher

criticality level. We shall therefore focus on addressing the remaining two issues.

As described in [TSP11, EN16, Wik16a], conventionally, the industry favors strict (spatial and temporal) isolations among different criticality levels. This is because isolation is often suggested by safety standards for the ease of certification [do11, iso11] – with rigorous isolation, guarantees made on different criticality levels (or for different applications) are independent of each other; this enables the independent certifications of different criticality levels (or applications), greatly reducing the certification burden as otherwise the system has to be repeatedly certified as a whole. Furthermore, it is up to the individual criticality level to adopt its own model to characterize the desired guarantee. Note, however, that strict isolation does not come for "free" – it usually tends to under utilize system resources [BD16] as the partitions are almost never fully utilized given the pessimism in WCET measurements.

Asymetrical Isolation (AIS): Current research commonly relaxes the strict isolation due to its resource inefficiency [BD16]. The first attempt is to "mix" tasks of different criticality levels on each core. However, this causes mutual interferences among all criticality levels and would require, in the worst-case, all tasks to be certified to the highest criticality level. Hence, an alternative approach is to asymmetrically isolate different criticality levels while retaining resource efficiency: WCETs of tasks are modeled on all criticality levels; whenever a task exceeds the WCET on one criticality level at runtime, only tasks with criticality levels greater than this level are guaranteed afterwards. Thus, whenever a time urgency happens, the system performs a runtime adaptation and only protects the critical tasks by reallocating resources from less critical tasks to them. Note that, the AIS model requires continuous runtime system monitoring to detect possible timing threats and to trigger the asymmetrical protection (we shall also call this approach as **implicit isolation** in the sequel).

Motivation & Contribution: The commonly assumed AIS model could incur drawbacks in that it can be abrupt and pessimistic. It is abrupt in the sense that whenever any critical task overruns, all less critical tasks immediately receive no services afterwards; further, it is pessimistic because, even if some critical tasks overrun, the system might still be able to provide services for less critical tasks. From another perspective, in a mixed-criticality setting, a "less" critical task can still be critical to the correct system functioning. As an example, consider a typical avionics use-case, where the fight control subsystem is of DO-178B [do11] criticality level A while the flight management subsystem is of criticality levels B and C (with A being the highest and E being the

lowest); if a level A task overruns, the flight management tasks cannot be simply abandoned as they are also crucial to the safety of the airplane. In light of this, meaningful and explicit guarantees for those tasks should be introduced to the AIS model. Moreover, the AIS model asymmetrically isolates different criticality levels only on the processing cores, neglecting other shared resources like memory. A new scheduling model would be required to cope with this. As a step towards addressing the above meaningful concerns for mixed-criticality systems, we make the following two main contributions.

1 We propose three different mixed-criticality models to address the limit of the AIS model regarding the service guarantees to less critical tasks. The first model is the service adaptation model which explicitly specifies the required service of less critical tasks in the urgent mode as well as the service resetting time from entering the urgent mode to resuming the nominal mode. The second model is the interference constraint graph (ICG), where for each task, the set of less critical tasks it can affect when overrunning is explicitly modeled as directed edges from this task to the affected tasks. We show that the AIS model is rather a special case of the ICG model; we further examine the properties of this new model and propose methods to reduce the interferences to less critical tasks by optimizing the ICG. Last, we present an extension of the AIS model augmented with processor speedup/overclocking. We show that speedup in situation of overrun can not only help to protect the timeliness of critical tasks, but also to improve the degraded services for less critical tasks. Furthermore, we show that speedup is even more attractive as it can help the system to recover faster to normal operation. For each of the proposed models, we present corresponding scheduling techniques based on fixed-priority (FP) or earliest deadline first (EDF) as well as their theoretical analyses.

2 We propose the global temporal isolation (IS) scheduling model for mixed-criticality systems, providing rigid isolation among criticality levels on the entire computing platform. In IS, the entire multi-core platform is viewed as a single resource, which is time partitioned among different criticality levels. In other words, IS enforces mutually exclusive execution among different criticality levels, avoiding inter level interference by construction. Within each time partition, the associated criticality level can assume its appropriate timing requirements, i.e., WCET measurements; any proper scheduling techniques can be adopted to exploit the multi-core platform and to optimize the resource utilization. Although

the concept of IS has been approached by several recent advances in mixed-criticality [GSHT13, BFB15], we are the first to formalize this model and formally study it. We propose optimal scheduling algorithms based on this model and theoretically analyze the limit (schedulability loss) of this model compared to when IS is not enforced. Furthermore, it is possible to combine the IS model with the AIS model to further increase system resource efficiency – if some task in one partition overruns its WCET on one criticality level, all subsequent partitions with criticality levels lower could be dropped or their services can be reconfigured or adapted.

Chapter Organization: We first briefly introduce the classical mixed-criticality models and corresponding scheduling techniques in Section 2.1. We then present three different mixed-criticality scheduling models (Section 2.2 to 2.4), all endeavoring to reduce the pessimisms of the state-of-the-art mixed-criticality task model. We then continue to formalize in Section 2.5 several recent advances of mixed-criticality scheduling into the isolation scheduling model and formally study the corresponding scheduling technique as well as limits of this model.

2.1 State-of-the-art Mixed-Criticality Models

In the classical mixed-criticality task model [BD16], a task set τ consists of independent sporadic tasks $\{\tau_1, \tau_2, ..., \tau_{|\tau|}\}$, which are to be scheduled on a platform with identical preemptive processors $\pi = \{\pi_1, \pi_2, ..., \pi_{|\pi|}\}$ ($|\pi| = 1$ in case of a single-core). Each task τ_i is specified by a minimum inter-arrival time T_i, a deadline D_i, and an associated criticality level of χ_i. Often, it is assumed that tasks have implicit deadlines [BBD+12] ($\forall \tau_i$, $D_i = T_i$) and dual-criticality levels ($\forall \tau_i$, χ_i can be either high (HI) or low (LO)). Unless otherwise stated, we will assume such an implicit deadline dual-criticality task model. In addition, to ensure timing safety, worst-case execution time (WCET) estimations of HI criticality tasks are more conservative than those of LO criticality tasks.

To further improve resource efficiency, the state-of-the-art mixed-criticality model [BD16] assumes to measure task WCETs on all criticality levels. Any HI criticality task τ_i has a LO criticality WCET C_i(LO) and a more pessimistic HI criticality WCET C_i(HI). Any LO criticality task τ_i has only a LO criticality WCET C_i(LO) and is not allowed to overrun C_i(LO). At runtime, the system starts with LO operation mode and guarantees deadlines for all HI and LO criticality tasks, assuming their LO criticality WCETs. If any HI criticality task overruns its LO criticality WCET, then the system switches immediately to HI operation mode and

all LO criticality tasks are dropped in order to guarantee HI criticality tasks. However, the system can switch back to LO mode at any time when there are no pending tasks [SGTG12]. Since such a mixed-criticality model effectively separates the guarantees among different criticality levels through runtime monitoring and reconfiguration, we shall call it the Asymmetric Isolation (AIS) model. From the above discussion, it is also not hard to see that such a model can be easily extended to more than two criticality levels by requiring WCET measurements on all those levels and by triggering a mode switch whenever a task exceeds the WCET on its own criticality level.

Under the AIS model, a task set is said *schedulable* if there exists a scheduling technique, such that all tasks can meet their deadlines in LO mode adhering to their LO level WCETs, and all HI criticality tasks can meet their deadlines in HI mode with HI criticality WCETs. Moreover, we say that a task set is schedulable on a processor with a speedup factor $s > 0$, if the same task set with all task WCETs divided by s is schedulable on the original processor.

Note that, the industrial best practice (e.g., see the avionics standard ARINC 653 [Wik16a]) still follows strict spatial and/or temporal isolation for different criticality levels – the asymmetric protection of different criticality levels is not required, as adopted in the state-of-the-art model in the research community.

For notational convenience, we define $U_{\chi_1}^{\chi_2}$ for $\chi_1, \chi_2 \in \{\text{LO}, \text{HI}\}$ as follows:

$$U_{\chi_1}^{\chi_2} = \sum_{\tau_i \in \tau \wedge \chi_i = \chi_1} \frac{C_i(\chi_2)}{T_i}.$$

$U_{\chi_1}^{\chi_2}$ denotes the total utilization of all χ_1 criticality tasks with their χ_2 criticality WCETs. For instance, $U_{\text{HI}}^{\text{LO}}$ denotes the utilization of HI criticality tasks with their LO criticality WCETs. We further define τ_χ as the set of all χ criticality tasks, where $\chi \in \{\text{LO}, \text{HI}\}$, and use $[\![a]\!]^c$ to represent $\min(a, c)$.

2.1.1 Mixed-Criticality Scheduling

We provide a short overview of several mixed-criticality scheduling policies under the AIS model, as we will build upon them. For a more extensive overview, we refer to [BD16] as an excellent survey. We also provide a quick introduction to mixed-criticality scheduling approaches based on explicit isolation among criticality levels.

2.1.1.1 EDF Based Approach

For mixed-criticality scheduling on multi-core platforms, either global scheduling [LB12] or partitioned scheduling [KAZ11] can be applied. In this chapter, we focus on the latter since it is more common in industrial embedded systems. Mixed-criticality scheduling is strongly *NP*-hard even for simple task models on a uniprocessor [KAZ11]. Hence, heuristic algorithms with analytical bounds are often proposed. In this category, a well known approach is the partitioned EDF-VD scheduling [Bar14, GGDY14]. Here, HI criticality tasks are first mapped to all processors followed by mapping of LO criticality tasks. Different bin packing techniques can be adopted for task mapping, while system utilization bounds are enforced on all cores to obtain a feasible schedule [Bar14, GGDY14].

Subsequently, EDF-VD [BBD+12] scheduling is adopted on each processor. In EDF-VD, deadlines of all HI criticality tasks are down-scaled by a multiplication factor x ($0 < x \leq 1$) in LO mode to prioritize their executions. This will leave enough time before their actual deadlines to accommodate extra workload (overrun). Intuitively, a smaller x increases system utilization in LO mode but decreases system utilization in HI mode, as more jobs are completed in LO mode. As a result, it affects scheduling in both LO and HI modes. For a task set τ *running on a single-core*, a feasible range of x exists [BBD+12] (summarized as follows).

Theorem 2.1. *To guarantee system schedulability on a single-core under EDF-VD, x must be set in the following range:*

$$0 < \frac{U_{\mathrm{HI}}^{\mathrm{LO}}}{1 - U_{\mathrm{LO}}^{\mathrm{LO}}} \leq x \leq \left[\!\left[\frac{1 - U_{\mathrm{HI}}^{\mathrm{HI}}}{U_{\mathrm{LO}}^{\mathrm{LO}}} \right]\!\right]^{1}.$$

Proof. This follows from Theorem 1 and Theorem 2 in [BBD+12]. □

Essentially, this theorem states that there is a lower bound on x, below which LO mode will not be schedulable. Similarly, an upper bound exists, beyond which HI mode will not be schedulable.

2.1.1.2 Fixed Priority Based Approach

Another common approach to schedule traditional real-time systems is to employ fixed-priority scheduling [LL73, TB94], where each task receives a fixed priority offline and higher priority tasks are scheduled preferably in comparison to lower priority ones. Established methods

including response-time analysis and real-time calculus [TBW95, TCN00] offer efficient schedulability analysis for such systems. In the mixed-criticality context, fixed-priority scheduling and analysis have been extended to consider multiple criticality levels and asymmetrical isolation among them [BBD11b, Pat12]. One such important result (with slight reformulation) was given in [BBD11b]:

Theorem 2.2. *For a dual-criticality task set τ with constrained deadlines (i.e., $D_i \leq T_i, \forall \tau_i$) under fixed-priority preemptive scheduling, the response time, R_i, of task τ_i is:*

$$R_i = C_i(\chi_i) + \sum_{\tau_j \in \boldsymbol{hp}(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot \min\left\{C_j(\chi_i), C_j(\chi_j)\right\}, \tag{2.1}$$

where $\boldsymbol{hp}(i)$ denotes tasks with priorities higher than τ_i.

More sophisticated analyses have also been derived in [BBD11b] to tighten task response times and in [Pat12] to consider multi-core platforms.

2.1.1.3 Fluid Based Approach

[LPG+14] extended a well known optimal multi-core scheduling technique, DP-Fair [LFS+10], to dual-criticality task systems by proposing MC-Fluid. DP-Fair is a fluid based scheduling technique, which enforces proportional progress of all tasks within dedicated system slices. DP-Fair only requires a minimal set of rules for achieving proportional progress, covering many other scheduling techniques as special cases, e.g., the original P-Fair algorithm [BCPV93]. Conceptually similar to EDF-VD, MC-Fluid runs the system in two modes, with the deadlines of HI criticality tasks shortened in LO mode. In both LO and HI modes, DP-Fair is adopted. [LPG+14] provided a corresponding schedulability test and investigated the theoretical performance of MC-Fluid.

2.1.1.4 Approaches with Explicit Task Class/Criticality Isolation

Initial research on isolating task classes on multi-core platforms did not explicitly address interference between task classes when jobs concurrently access shared resources, i.e., they imply that it can be bounded. For instance, [ABB09] and [MEA+10] adopt different strategies (partitioned EDF, global EDF, cyclic executive) for different task classes and use a bandwidth reservation server for timing isolation between classes. However, interference analysis for multiple shared resources is a very challenging task by itself. In fact, estimating response time bounds

under contention may be even impossible for MC systems, because a certification authority for higher criticality tasks does not necessarily possess information on the behavior of lower criticality tasks that are co-hosted on the same platform.

Subsequently, researchers acknowledged the problem of inter-class interference and proposed mechanisms for criticality-aware arbitration of shared resources, with the objective of statically bounding interference from lower to higher criticality tasks. [YYP+12] and [FLY14] proposed a software-based memory throttling mechanism (with predefined [YYP+12] or dynamically allocated [FLY14] per-core budgets) to explicitly control interference on a shared memory controller. [PQnC+09, GAG13] proposed hardware modifications to a shared memory controller for mixed hard and soft real-time systems. Recent works [RLP+11, WKP13, YMWP14, GSHT14] proposed partitioning data to disjoint DRAM banks in order to minimize inter-core interference. [TSP15] presented optimization methods for time-triggered partition scheduling on heterogeneous multi-core that comply with the ARINC-653 standard [ARI03]; they assume that the platform provides both spatial and temporal partitioning that enforce enough isolation between task classes. Similarly, [KYBS14] relied on ARINC-653 compliance to devise a method for conflict-free I/O transactions. Finally, [TAED13] implemented virtualization and monitoring mechanisms to provide independence among flows of different criticality in networks-on-chips. These mechanisms and policies ensure sufficient isolation among criticality levels, but they suffer from poor flexibility, e.g., memory budgets [YYP+12] cannot change dynamically if the resource demand changes, and they may require special hardware support, which is not widely available.

Finally, [GSHT13] and [BFB15] recently proposed scheduling strategies that sidestep the need for fine-grained shared resource arbitration. The key idea is to only permit tasks of the same criticality (i.e., from the same class) to execute concurrently. Based on this insight, the scheduling policies they propose avoid resource interference among task classes, exploit multiple cores, and only suffer a limited schedulability loss to enforce time-partitioning among task classes. The Isolation Scheduling model we will propose includes both policies as special cases; additionally, we will propose novel policies built from scratch.

2.2 Mixed-Criticality Service Adaptation

As already discussed at the beginning of this chapter, it is often too pessimistic to reject all less critical tasks whenever a single high criticality

task exceeds at runtime a certain execution time threshold. Indeed, this has already been confirmed in [SGTG12, SZ13]. Here, instead of dropping less critical tasks completely, either partial dropping or best-effort scheduling are adopted for the less critical tasks. However, the service that the system can still guarantee to the less critical tasks when a critical one overruns is unknown. It is important to compute bounds on the service that can be provided for the less critical tasks, which remained as an open problem when we wrote the paper [HGST14]. Such information could be used to guide the online reconfiguration of the provided system services.

Furthermore, the guarantees made by the above scheduling techniques [SGTG12, SZ13] are rarely acceptable in practice. Recall the avionic use-case – the Flight Management System (FMS) application. If a localization task (certified at DO-178B criticality level B) exceeds a certain execution time threshold, we cannot simply reject the level C tasks (among them be the flightplan task) or perform best-effort scheduling for them, since the airplane constantly requires the flightplan information. Instead, a *degraded service* for the level C tasks should be guaranteed. Moreover, for FMS, when the system can be *reset* to provide the original service to all tasks is an important certification criteria. Therefore, offline analysis techniques need to be developed to bound such resetting time.

In the following, we will extend the original mixed-criticality scheduling model and the EDF-VD [BBD+12] scheduling technique to guarantee a degraded service for LO criticality tasks in the urgent scenario. We will further provide a bound on the service resetting time. To this end, we will extend the demand bound [EY12] and the arrival bound [TCN00] techniques to analyze this new type of systems. With our theoretical results, we demonstrate that a trade-off exists between the degraded service and the service resetting time. Finally, we validate our proposed techniques with a case study of an industrial application – the flight management system.

2.2.1 System Model

Our model differs from the commonly assumed mixed-criticality model in that, rather than dropping all LO criticality tasks in HI operation mode, we still guarantee a degraded service for all those tasks (i.e., decreased task activation frequencies and relaxed task deadlines). Without loss of generality, we can then abstract each task τ_i with a tuple, $\Big\{\big\{T_i(\text{HI}), C_i(\text{HI}), D_i(\text{HI})\big\}, \big\{T_i(\text{LO}), C_i(\text{LO}), D_i(\text{LO})\big\}, \chi_i\Big\}$:

- $T_i(\chi) \in \mathbb{R}^+$ is the minimum inter-arrival time of τ_i in mode χ.

- $C_i(\chi) \in \mathbb{R}^+$ is the WCET of τ_i in mode χ.
- $D_i(\chi) \in \mathbb{R}^+$ is the relative deadline of τ_i in mode χ.
- $\boldsymbol{\chi} = \{\text{HI}, \text{LO}\}$ is the set of criticality levels / operating modes.
- $\chi_i \in \boldsymbol{\chi}$ is the criticality level of task τ_i.

$\left\{T_i(\chi), C_i(\chi), D_i(\chi)\right\}$ characterizes the service to be guaranteed for τ_i in mode χ. A dual-criticality task set is said *schedulable* if there exists a scheduling technique, such that the service requirements for all tasks in both operating modes can be satisfied. For a HI criticality task τ_i, its WCET is adjusted from $C_i(\text{LO})$ to $C_i(\text{HI})$ when transiting to HI criticality mode. This can lead to a unschedulable system: Consider a scenario when a job of this task just finishes its low criticality WCET at its deadline, and at the same time a mode transition happens; this implies that this task must finish $C_i(\text{HI}) - C_i(\text{LO})$ units of execution in zero time. In order to schedule HI criticality tasks, their deadlines need to be tuned in LO criticality mode ($D_i(\text{LO}) \leq D_i(\text{HI})$), such that they can still meet their deadlines when transiting to HI criticality mode. This concept has been explored in [BBD+11a, BBD+12, EY12].

In summary, the following assumptions are made: If $\chi_i = \text{HI}$, then

$$T_i(\text{LO}) = T_i(\text{HI}) = T_i, C_i(\text{HI}) \geq C_i(\text{LO}), D_i(\text{LO}) \leq D_i(\text{HI}) = D_i; \tag{2.2}$$

if $\chi_i = \text{LO}$, then

$$T_i(\text{LO}) = T_i, T_i(\text{LO}) \leq T_i(\text{HI}), C_i(\text{HI}) = C_i(\text{LO}), D_i = D_i(\text{LO}) \leq D_i(\text{HI}). \tag{2.3}$$

Note that we do not pose any constraint on task deadlines with respect to their periods here.

For the original EDF-VD scheduling technique [BBD+11a, BBD+12], all tasks are scheduled by Earliest Deadline First (EDF) in both HI and LO criticality modes (i.e., *mode-switched EDF*). All LO criticality tasks are immediately dropped when the system transits to HI criticality mode. This can be viewed as a special case by setting $T_i(\text{HI}) := D_i(\text{HI}) := \infty$ for all LO criticality tasks. Furthermore, recall that, for notational convenience, we denote by τ_{HI} (τ_{LO}) the set of HI (LO) criticality tasks.

2.2.2 Mixed-Criticality Service Reconfiguration

We present in this section the analysis of dual-criticality task sets scheduled by mode-switched EDF. We first quantify the demand bounds of *all* tasks in LO and HI criticality modes, and present the corresponding schedulability test. We then present results on approximations of the

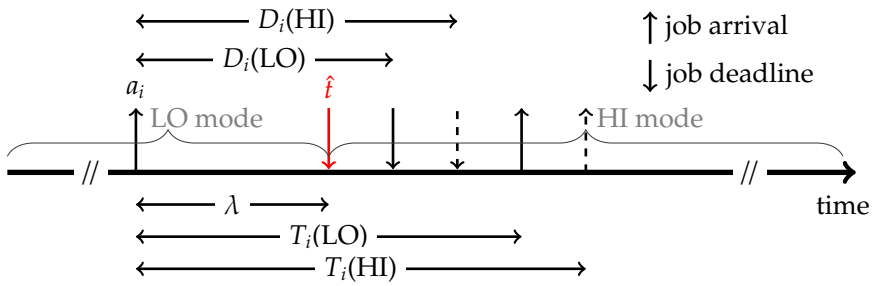


Figure 2.1: A mode transition triggered by task overrun

analyzed demand bounds. We view the service degradation of LO criticality tasks by a scaling factor (≥ 1) of their periods and deadlines. For implicit-deadline task sets ($T_i = D_i$), an algorithm is proposed in Section 2.2.2.3 to adjust the services of LO criticality tasks in HI mode. The results presented extend existing works in [BBD+11a, EY12, EY13].

2.2.2.1 Mixed-Criticality Demand Bound Analysis

The demand bound of a task in a given interval is defined as the sum of execution times of all task instances, which have arrival times and deadlines both in this interval. During LO criticality mode, the demand bound $\mathrm{dbf}_{\mathrm{LO}}$ of any task in an interval of length Δ can be bounded according to known results in [EY12]:

$$\mathrm{dbf}_{\mathrm{LO}}(\tau_i, \Delta) = \max\left\{\left\lfloor \frac{\Delta - D_i(\mathrm{LO})}{T_i(\mathrm{LO})} \right\rfloor + 1, 0\right\} \cdot C_i(\mathrm{LO}). \tag{2.4}$$

For HI criticality mode, we need to consider the impact of unfinished jobs at the transition point. Those are the jobs for which the schedules may be changed (i.e., tasks are scheduled by EDF and their deadlines may be adjusted at the time of mode switch).

We depict in Figure 2.1 a system undergoing a mode transition at time $\hat{t}$. A job of task τ_i arrives in LO mode at time a_i. A mode transition is signaled λ units after that. From this time on, all jobs of this task are guaranteed with HI criticality parameters.

In [EY12], only the demand bounds of HI criticality tasks in HI criticality mode are derived. This is done by identifying a worst-case demand bound including that of the unfinished job at the transition time, see equation (2.5). We proceed to present a general approach to analyze the demand bounds of *all* tasks in HI criticality mode. Let us further define a set of functions (2.6)-(2.9):

$$\mathrm{RM}(\tau_i, \lambda) = C_i(\mathrm{HI}) - C_i(\mathrm{LO}) + \min\left\{D_i(\mathrm{LO}) - \lambda, C_i(\mathrm{LO})\right\}, \tag{2.5}$$

$$\mathrm{dbf}^1_{\mathrm{HI}}(\tau_i, \Delta) = \max\left\{\left\lfloor \frac{\Delta - D_i(\mathrm{HI})}{T_i(\mathrm{HI})} \right\rfloor + 1, 0\right\} \cdot C_i(\mathrm{HI}), \tag{2.6}$$

$$\mathrm{dbf}_{\mathrm{RM}}(\tau_i, \lambda, \Delta) = \begin{cases} \mathrm{RM}(\tau_i, \lambda) & \text{if } \Delta \geq D_i(\mathrm{HI}) - \lambda, \quad (2.7a) \\ 0 & \text{if } \Delta < D_i(\mathrm{HI}) - \lambda. \quad (2.7b) \end{cases}$$

$$\begin{aligned} \mathrm{dbf}^2_{\mathrm{HI}}(\tau_i, \lambda, \Delta) &= \mathrm{dbf}_{\mathrm{RM}}(\tau_i, \lambda, \Delta) \\ &+ \max\left\{\left\lfloor \frac{\Delta - D_i(\mathrm{HI}) - (T_i(\mathrm{HI}) - \lambda)}{T_i(\mathrm{HI})} \right\rfloor + 1, 0\right\} \cdot C_i(\mathrm{HI}). \end{aligned} \tag{2.8}$$

$$\mathrm{dbf}_{\mathrm{HI}}(\tau_i, \Delta) = \sup\left\{\mathrm{dbf}^1_{\mathrm{HI}}(\tau_i, \Delta), \sup_{0 \leq \lambda \leq D_i(\mathrm{LO})} \left\{\mathrm{dbf}^2_{\mathrm{HI}}(\tau_i, \lambda, \Delta)\right\}\right\}. \tag{2.9}$$

Formally, we have the following result.

Lemma 2.1. *The demand bound function of any task in* HI *criticality mode can be calculated by equation (2.9).*

Proof. Let us consider two different cases.

1- $\lambda \in [0, D_i(\mathrm{LO})]$: In this case, the current job of τ_i may have not finished its LO criticality WCET, the worst-case left-overs of the current job can be bounded by $\min\{D_i(\mathrm{LO}) - \lambda, C_i(\mathrm{LO})\}$. Since starting from $\hat{t}$ the current and future jobs of τ_i execute according to HI criticality mode parameters, the total left-overs of the current job of τ_i can be bounded by equation (2.5). We have to further identify a time interval of length Δ within $[\hat{t}, +\infty)$, which has the worst-case demand bound. Suppose that such an interval is represented as $[\hat{t}+\eta, \hat{t}+\eta+\Delta]$ ($\eta \geq 0$), i.e., the interval starts η units of time after $\hat{t}$. Let us consider further two different cases:

- $\eta > 0$: In this case, the left-over job does *not* belong to the interval and its demand is not considered. Let γ represent the value of $((\lambda + \eta) \mod T_i(\mathrm{HI}))$, then the number of maximum complete arrivals of τ_i in $[\hat{t} + \eta, \hat{t} + \eta + \Delta]$ is bounded by:

$$\begin{cases} \max\left\{\left\lfloor \frac{\Delta - D_i(\mathrm{HI}) - (T_i(\mathrm{HI}) - \gamma)}{T_i(\mathrm{HI})} \right\rfloor + 1, 0\right\} & \text{if } 0 < \gamma, \quad (2.10a) \\ \max\left\{\left\lfloor \frac{\Delta - D_i(\mathrm{HI})}{T_i(\mathrm{HI})} \right\rfloor + 1, 0\right\} & \text{if } \gamma = 0. \quad (2.10b) \end{cases}$$

Hence, when $\gamma = 0$ we can get the maximum complete arrivals of τ_i within such an interval. And the demand bound of τ_i in $[\hat{t} + \eta, \hat{t} + \eta + \Delta]$ is upper bounded by equation (2.6).

• $\eta = 0$: In this case the left-over job needs to be considered for the demands in interval $[\hat{t}, \hat{t} + \Delta]$. Notice further that the left-over job is only considered when $\Delta \geq D_i(\text{HI}) - \lambda$, hence the demand of this job is given by equation (2.5). The maximum number of future complete arrivals of τ_i within$[\hat{t}, \hat{t} + \Delta]$ is bounded by:

$$\max\left\{\left\lfloor \frac{\Delta - D_i(\text{HI}) - (T_i(\text{HI}) - \lambda)}{T_i(\text{HI})} \right\rfloor + 1, 0\right\}. \tag{2.11}$$

Hence the demand bound function of τ_i can be given by equation (2.8).

2- $\lambda \in (D_i(\text{LO}), T_i(\text{LO})]$: In this case τ_i has no active instance running in the system, we can derive similarly that the maximum number of complete arrivals of τ_i within an interval of length Δ happens when this interval starts with an arrival of τ_i. This number is again bounded by equation (2.10b). Hence the demand bound function in this case can be represented by equation (2.6).

Now, the demand bound function for any task τ_i in HI criticality mode can be represented by equation (2.9). □

Based on the above computed demand bounds, schedulability of a task set can be tested using existing results.

Theorem 2.3. *[EY12] A dual-criticality task set τ is schedulable on a unit-speed processor, if $\forall \Delta \geq 0$:*

$$\max\left\{\sum_{\tau} \text{dbf}_{\text{LO}}(\tau_i, \Delta), \sum_{\tau} \text{dbf}_{\text{HI}}(\tau_i, \Delta)\right\} \leq \Delta. \tag{2.12}$$

2.2.2.2 Mixed-Criticality Demand Bound Approximation

For the original EDF-VD scheduling technique, all LO criticality tasks are rejected in HI criticality mode. The problem we are addressing in this work is different, as we aim at computing the bounds on the service for LO criticality tasks in HI criticality mode. This is not a trivial problem as both HI and LO criticality tasks change their schedules immediately when transiting to HI criticality mode, which makes the analysis difficult. In order to simplify the problem, we restrict ourselves to consider implicit-deadline task sets ($D_i = T_i$). According to the discussions in Section 2.2.1, the deadlines of HI criticality tasks need to be tuned in LO criticality mode in order to guarantee their deadlines in HI criticality mode. Similar to

the original EDF-VD scheduling technique, we assume that the deadline tuning for HI criticality tasks is characterized by a scaling factor x ($x \leq 1$):

$$T_i(\text{LO}) = T_i(\text{HI}) = D_i(\text{HI}), D_i(\text{LO}) = xD_i(\text{HI}). \tag{2.13}$$

In addition, we assume that the service degradation of LO criticality tasks is characterized by another scaling factor y ($y \geq 1$):

$$T_i(\text{LO}) = D_i(\text{LO}), T_i(\text{HI}) = D_i(\text{HI}) = yT_i(\text{LO}). \tag{2.14}$$

We proceed to show that the demand bounds of all tasks in HI criticality mode (Lemma 2.7) can be tightly approximated based on equation (2.9). The essential idea is to bound the nonlinear function $\text{dbf}_{\text{HI}}(\tau_i, \Delta)$ by certain slopes, for both HI criticality and LO criticality tasks. The following results are presented.

Lemma 2.2. *For a* HI *criticality task* τ_i*,*

$$\text{dbf}_{\text{HI}}(\tau_i, \Delta) \leq \max\left\{\frac{C_i(\text{HI}) - C_i(\text{LO})}{(1-x)T_i}, \frac{C_i(\text{HI})}{C_i(\text{LO}) + (1-x)T_i}\right\} \cdot \Delta. \tag{2.15}$$

Proof. Let us consider two different cases.

1- $0 \leq \Delta \leq T_i$: According to equations (2.6), (2.8), (2.9), we can derive:

- $0 \leq \Delta < (1-x)T_i$: In this case, both $\text{dbf}^1_{\text{HI}}(\tau_i, \Delta)$ and $\text{dbf}^2_{\text{HI}}(\tau_i, \lambda, \Delta)$ (independent of λ) evaluate to zero. Hence, the demand bound is constantly zero.
- $(1-x)T_i \leq \Delta \leq C_i(\text{LO}) + (1-x)T_i$: In this case, $\text{dbf}^1_{\text{HI}}(\tau_i, \Delta)$ evaluates to zero. $\text{dbf}^2_{\text{HI}}(\tau_i, \lambda, \Delta)$ has the maximum value $C_i(\text{HI}) - C_i(\text{LO}) + \Delta - (1-x)T_i$ when $\lambda = T_i - \Delta$. Hence, the worst-case demand bound is $C_i(\text{HI}) - C_i(\text{LO}) + \Delta - (1-x)T_i$.
- $C_i(\text{LO}) + (1-x)T_i \leq \Delta \leq T_i$: In this case, $\text{dbf}^1_{\text{HI}}(\tau_i, \Delta)$ evaluates to $C_i(\text{HI})$. The worst-case of $\text{dbf}^2_{\text{HI}}(\tau_i, \lambda, \Delta)$ also evaluates to $C_i(\text{HI})$ (when $\lambda \leq xT_i - C_i(\text{LO})$). Hence, the maximum demand bound is $C_i(\text{HI})$.

2- $(k-1)T_i \leq \Delta \leq kT_i$ ($k \in \mathbb{N}^+$): Assume that $\Delta = (k-1)T_i + \delta$ ($0 \leq \delta \leq T_i$). First, we can derive that:

$$\text{dbf}^1_{\text{HI}}(\tau_i, \Delta) = (k-1)C_i(\text{HI}) + \text{dbf}^1_{\text{HI}}(\tau_i, \delta). \tag{2.16}$$

Second, $\text{dbf}^2_{\text{HI}}(\tau_i, \lambda, \Delta)\}$ achieves the maximum value when $\lambda = T_i - \delta$,

then:

$$\begin{aligned}
&\sup_{0\le\lambda\le xT_i}\left\{\mathrm{dbf}^2_{\mathrm{HI}}(\tau_i,\lambda,(k-1)T_i+\delta)\right\}\\
&=\mathrm{dbf}^2_{\mathrm{HI}}(\tau_i,T_i-\delta,(k-1)T_i+\delta)\\
&=(k-1)C_i+\mathrm{dbf}^2_{\mathrm{HI}}(\tau_i,T_i-\delta,\delta)\\
&=(k-1)C_i+\sup_{0\le\lambda\le xT_i}\{\mathrm{dbf}^2_{\mathrm{HI}}(\tau_i,\lambda,\delta)\}.
\end{aligned} \tag{2.17}$$

Hence,

$$\begin{aligned}
\mathrm{dbf}_{\mathrm{HI}}(\tau_i,\Delta)&=\sup\left\{\mathrm{dbf}^1_{\mathrm{HI}}(\tau_i,\Delta),\ \sup_{0\le\lambda\le xT_i}\left\{\mathrm{dbf}^2_{\mathrm{HI}}(\tau_i,\lambda,\Delta)\right\}\right\}\\
&=(k-1)C_i(\mathrm{HI})+\mathrm{dbf}_{\mathrm{HI}}(\tau_i,\delta).
\end{aligned} \tag{2.18}$$

□

Lemma 2.3. *For a* LO *criticality task* τ_i,

$$\mathrm{dbf}_{\mathrm{HI}}(\tau_i,\Delta)\le\frac{C_i(\mathrm{LO})}{C_i(\mathrm{LO})+(y-1)T_i}\cdot\Delta. \tag{2.19}$$

Proof. This can be similarly derived as shown in Lemma 2.2. □

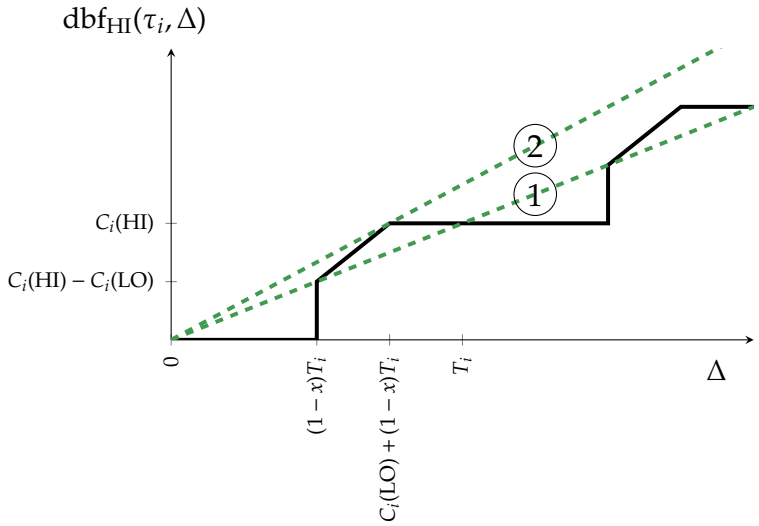
We plot now in Figure 2.2(a) and Figure 2.2(b) the approximations of demand bounds in HI criticality mode for both HI and LO criticality tasks (dashed lines).

2.2.2.3 Mixed-Criticality Service Reconfiguration

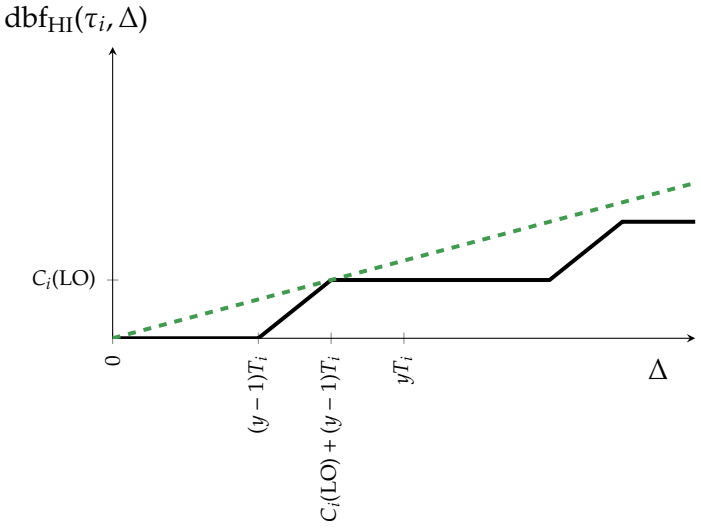
We now show how to reconfigure the system such that a maximum degraded service for LO criticality tasks can be guaranteed. Our analysis is based on the demand bound approximations shown in the previous section. For notational convenience, we define two functions as follows:

$$\begin{aligned}
h(x)&=\sum_{\tau_{\mathrm{HI}}}\frac{U_i(\mathrm{HI})}{U_i(\mathrm{LO})+(1-x)},\\
l(y)&=\sum_{\tau_{\mathrm{LO}}}\frac{U_i(\mathrm{LO})}{U_i(\mathrm{LO})+(y-1)},
\end{aligned} \tag{2.20}$$

where $U_i(\chi)=C_i(\chi)/T_i$. $h(x)$ and $l(y)$ represent the summed slopes of HI and LO criticality tasks in HI criticality mode, respectively. We further use $U^{\chi_2}_{\chi_1}$ to denote $\sum_{\tau_i:\chi_i=\chi_1}U_i(\chi_2)$.



(a) Approximation of $\mathrm{dbf}_{\mathrm{HI}}(\tau_i, \Delta), \chi_i = \mathrm{HI}$, see calcualtion (2.15) (note that slope 1 will be above slope 2 if $\frac{C_i(\mathrm{HI})-C_i(\mathrm{LO})}{(1-x)T_i} > 1$)



(b) Approximation of $\mathrm{dbf}_{\mathrm{HI}}(\tau_i, \Delta), \chi_i = \mathrm{LO}$, see calculation (2.19)

Figure 2.2: Approximation of HI mode system demand bound function

We now explain the service reconfiguration in Algorithm 2.1. For the case when $U_{\mathrm{HI}}^{\mathrm{HI}} + U_{\mathrm{LO}}^{\mathrm{LO}} \leq 1$, LO criticality tasks need not be reconfigured as the system can guarantee all tasks with their worst-case service requirements. For the case when $U_{\mathrm{HI}}^{\mathrm{LO}} + U_{\mathrm{LO}}^{\mathrm{LO}} > 1$, the system is even not schedulable during LO criticality mode. Excluding the above conditions, the system needs to be further tested to see whether the reconfiguration of services in HI criticality mode is feasible. This is ensured by enforcing that

Algorithm 2.1: Service reconfiguration

```
   Input: τ
1  if U_HI^HI + U_LO^LO ≤ 1 then
2  |  x ← 1;
3  |  y ← 1;
4  else
5  |  if U_HI^LO + U_LO^LO ≤ 1 then
6  |  |  x ← U_HI^LO / (1 − U_LO^LO);
7  |  |  if h(x) ≤ 1 then
8  |  |  |  y ← inf{y ≥ 1 : h(x) + l(y) ≤ 1};
9  |  |  else
10 |  |  |  return false;
11 |  |  end
12 |  else
13 |  |  return false;
14 |  end
15 end
16 return true;
```

the total slopes of tasks in HI criticality mode is ≤ 1 (line 8). A minimal service degradation factor y, that satisfies this condition, is computed based on equation (2.20).

Formally, we have the following result.

Theorem 2.4. *Given a dual-criticality task set, Algorithm 2.1 can compute a minimized service degradation factor y for* LO *criticality tasks, and a corresponding deadline tuning factor x for* HI *criticality tasks, such that the task set is schedulable in all operating modes.*

Proof. The schedulability of a task set in LO criticality mode is tested by line 5 in Algorithm 2.1. We have to enforce further the schedulability in HI mode according to Theorem 2.3. According to Lemma 2.2 and Lemma 2.3:

$$\begin{aligned}
&\sum_{\tau} \mathrm{dbf}_{\mathrm{HI}}(\tau_i, \Delta) \leq \Delta \\
\Leftarrow &\sum_{\tau_{\mathrm{HI}}} \max\left\{\frac{C_i(\mathrm{HI}) - C_i(\mathrm{LO})}{(1-x)T_i}, \frac{C_i(\mathrm{HI})}{C_i(\mathrm{LO}) + (1-x)T_i}\right\} \\
&+ \sum_{\tau_{\mathrm{LO}}} \frac{C_i(\mathrm{LO})}{C_i(\mathrm{LO}) + (y-1)T_i} \leq 1 \\
\Leftrightarrow &h(x) + l(y) \leq 1.
\end{aligned} \tag{2.21}$$

Table 2.1: Example 2.1 task set

τ	τ_1	τ_2	τ_3	τ_4	τ_5
χ	HI	LO	LO	LO	LO
T/D	60	8	30	90	15
C(HI)	18	4	4	6	3
C(LO)	3	4	4	6	3

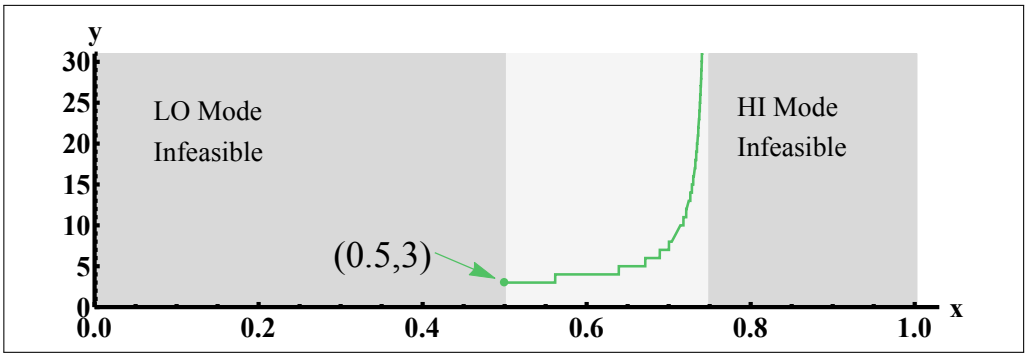


Figure 2.3: Trade off between x and y

If all LO criticality tasks are rejected in HI criticality mode, the inequality becomes $h(x) \leq 1$, which is a sufficient condition for all HI criticality tasks to be schedulable in HI criticality mode by rejecting all LO criticality tasks. Furthermore, we observe that the minimal value of y increases with increasing x. Hence, we can set x to $\frac{U_{\text{HI}}^{\text{LO}}}{1-U_{\text{LO}}^{\text{LO}}}$ (the minimum x to guarantee schedulability in LO criticality mode) in order to get the maximum degraded service for LO criticality tasks (minimal y): $y = \inf\{y \geq 1 : h(x) + l(y) \leq 1\}$. □

Example 2.1. *Consider the set of dual-criticality task set as shown in Table 2.1. According to Algorithm 2.1, we can derive* $x = 0.5,\ y = 2.6488$.

In practice, one can ceil the value of y *and multiply the periods of* LO *criticality tasks by* 3 *in* HI *criticality mode. This can be done by the scheduler skipping 2 task instances in every 3 arrivals of a task. Furthermore, according to Theorem 2.4, there is a trade off between* x *and* y. *The more we shorten the deadlines of* HI *criticality tasks (providing that all tasks are still schedulable in* LO *criticality mode), the better degraded service we can get for* LO *criticality tasks. We plot in Figure 2.3 the ceiling of* y *as a function of* x. *Notice that for* $x < 0.5$ *the system will not be schedulable in* LO *criticality mode and for* $x > 0.75$ *the system will not be schedulable in* HI *criticality mode even assuming all* LO *criticality tasks are rejected.*

2.2.3 Mixed-Criticality Service Resetting

We present in this section how to quantify the resetting time of the system services, i.e., the elapsed time between when the system transits from LO criticality mode to HI criticality mode and when it can safely transit back. This is not a trivial problem since we do not know offline the actual task execution times at runtime. Our analysis in this section will not assume any such information. First, a sufficient condition for resetting the system service is given as follows.

Lemma 2.4. *[SGTG12] A dual-criticality system can be safely reset to* LO *criticality mode if the processor is idle.*

Based on Lemma 2.4, a simple runtime mechanism can be implemented to reset the system to LO criticality mode. However, it would be particularly interesting to quantify statically the worst-case resetting time, as an important measure of the guarantees that a mixed-criticality scheduling algorithm can provide.

We use the same notion as shown in Figure 2.1, where a system transits to HI criticality mode at time $\hat{t}$. Furthermore, we define the arrival demand function of a task (adf) in the interval $[\hat{t}, \hat{t} + \Delta]$ as the cumulative execution times of all task instances *issued* within this interval. We define a list of functions as follows:

$$\mathrm{adf}^1_{\mathrm{HI}}(\tau_i, \lambda, \Delta) = \mathrm{RM}(\tau_i, \lambda) + \max\left\{\left\lceil \frac{\Delta - (T_i(\mathrm{HI}) - \lambda)}{T_i(\mathrm{HI})} \right\rceil, 0\right\} \cdot C_i(\mathrm{HI}), \quad (2.22)$$

$$\mathrm{adf}^2_{\mathrm{HI}}(\tau_i, \Delta) = \left\lceil \frac{\Delta}{T_i(\mathrm{HI})} \right\rceil \cdot C_i(\mathrm{HI}), \quad (2.23)$$

$$\mathrm{adf}_{\mathrm{HI}}(\tau_i, \Delta) = \sup\left\{\mathrm{adf}^2_{\mathrm{HI}}(\tau_i, \Delta), \sup_{0 \le \lambda \le D_i(\mathrm{LO})} \left\{\mathrm{adf}^1_{\mathrm{HI}}(\tau_i, \lambda, \Delta)\right\}\right\}. \quad (2.24)$$

Formally, we have the following results.

Lemma 2.5. *The arrival demand function of any task in the interval* $[\hat{t}, \hat{t} + \Delta]$ *can be calculated by equation (2.24).*

Proof. This lemma can be similarly derived as shown in the proof of Lemma 2.7.

1-$\lambda \in [0, D_i(\mathrm{LO})]$: In this case, the remaining execution demand of the current job needs to be counted, see equation (2.5). The number of future arrivals within $[\hat{t}, \hat{t} + \Delta]$ can be bounded by: $\max\left\{\left\lceil \frac{\Delta - (T_i(\mathrm{HI}) - \lambda)}{T_i(\mathrm{HI})} \right\rceil, 0\right\}$. Hence, the total arrived demands in $[t, t + \Delta]$ can be given by equation (2.22).

2-$\lambda \in (D_i(\text{LO}), T_i(\text{LO})]$: Clearly, in this case, there is no unfinished instance of τ_i in the system. We get the worst-case arrived demands within $[\hat{t}, \hat{t}+\Delta]$ when t coincides with an arrival of τ_i, which is bounded by equation (2.23).

Summarizing: the worst-case arrived demands within $[\hat{t}, \hat{t}+\Delta]$ is given by equation (2.24). □

Theorem 2.5. *The service of the system can be reset at time $\hat{t}+\Delta_R$, where Δ_R is lower bounded by $\frac{\sum_\tau C_i(\chi_i)}{1-h(x)-l(y)}$.*

Proof. We identify a processor idle time after $\hat{t}$, denoted as $\hat{t}+\Delta_\text{R}$, at which time the system can be safely reset to LO criticality mode. It then must be the case that $\sum_\tau \text{adf}_\text{HI}(\tau_i, \Delta_\text{R}) \leq \Delta_\text{R}$.

To show a lower bound of Δ_R, similar to Lemma 2.2 and Lemma 2.3, we can first approximate the arrival demand functions for both HI and LO criticality tasks in the interval $[\hat{t}, \hat{t}+\Delta]$:

$$\begin{aligned}
\text{adf}_\text{HI}(\tau_i, \Delta) &\leq C_i(\text{HI}) + \frac{C_i(\text{HI})}{C_i(\text{LO}) + (1-x)\times T_i} \times \Delta \quad \text{if } \chi_i = \text{HI}, \\
\text{adf}_\text{HI}(\tau_i, \Delta) &\leq C_i(\text{LO}) + \frac{C_i(\text{LO})}{C_i(\text{LO}) + (y-1)\times T_i} \times \Delta \quad \text{if } \chi_i = \text{LO}.
\end{aligned} \tag{2.25}$$

It follows that:

$$\begin{aligned}
&\sum_\tau \text{adf}_\text{HI}(\tau_i, \Delta_\text{R}) \leq \Delta_\text{R} \\
\Leftarrow &\sum_\tau C_i(\chi_i) + \left(\sum_{\tau_\text{HI}} \frac{C_i(\text{HI})}{C_i(\text{LO}) + (1-x)\times T_i} \right. \\
&\left. + \sum_{\tau_\text{LO}} \frac{C_i(\text{LO})}{C_i(\text{LO}) + (y-1)\times T_i} \right) \times \Delta_\text{R} \leq \Delta_\text{R} \\
\Leftrightarrow &\Delta_\text{R} \geq \frac{\sum_\tau C_i(\chi_i)}{1 - \frac{U_\text{HI}^\text{HI}}{1-x} - \frac{U_\text{LO}^\text{LO}}{y-1}}.
\end{aligned} \tag{2.26}$$

□

Notice that according to Theorem 2.5, if we decrease x, then the resetting time will be reduced. Hence, one can simply set x as $\frac{U_\text{HI}^\text{LO}}{1-U_\text{LO}^\text{LO}}$, which is the minimum to guarantee the schedulability of tasks in LO criticality mode. For setting y, there is a trade-off between the resetting time and

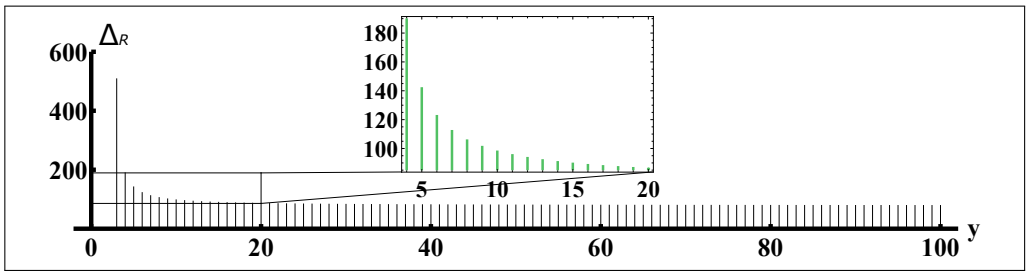


Figure 2.4: Trade off between Δ_R and y

Table 2.2: Task parameters for the FMS application in units of *ms*

τ	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6
T/D	5000	200	1000	1600	100	1000
C(LO)	$\{0, 20\}$	$\{0, 20\}$	$\{0, 20\}$	$\{0, 20\}$	$\{0, 20\}$	$\{0, 20\}$
χ	B	B	B	B	B	B
τ	τ_7	τ_8	τ_9	τ_{10}	τ_{11}	
T/D	1000	1000	1000	1000	1000	
C(LO)	$\{0, 20\}$	$\{0, 200\}$	$\{0, 200\}$	$\{0, 200\}$	$\{0, 200\}$	
χ	B	C	C	C	C	

the degraded service in HI criticality mode: if we increase the degraded service for LO criticality tasks (i.e., decrease y), then the resetting time will be increased.

Example 2.2. *Consider the same task set as shown in Example 2.1. We can now plot the service resetting time as a function of y, see Figure 2.4.*

As one can see, the resetting time decreases with increasing y. This gives us the flexibility to trade the degraded service of LO *criticality tasks for the resetting time. Furthermore, as y increases, the gain of saving in resetting time will also decrease.*

2.2.4 Case Study

In the European research project Certainty [cer], a Flight Management System (FMS) case study has been chosen and implemented. We validate in this section our approach with this avionics usecase; the system model we will use in our experiments follow the major properties of the FMS case study. We show the applicability of our approach, such that service requirements for different runtime modes can be guaranteed. In addition,

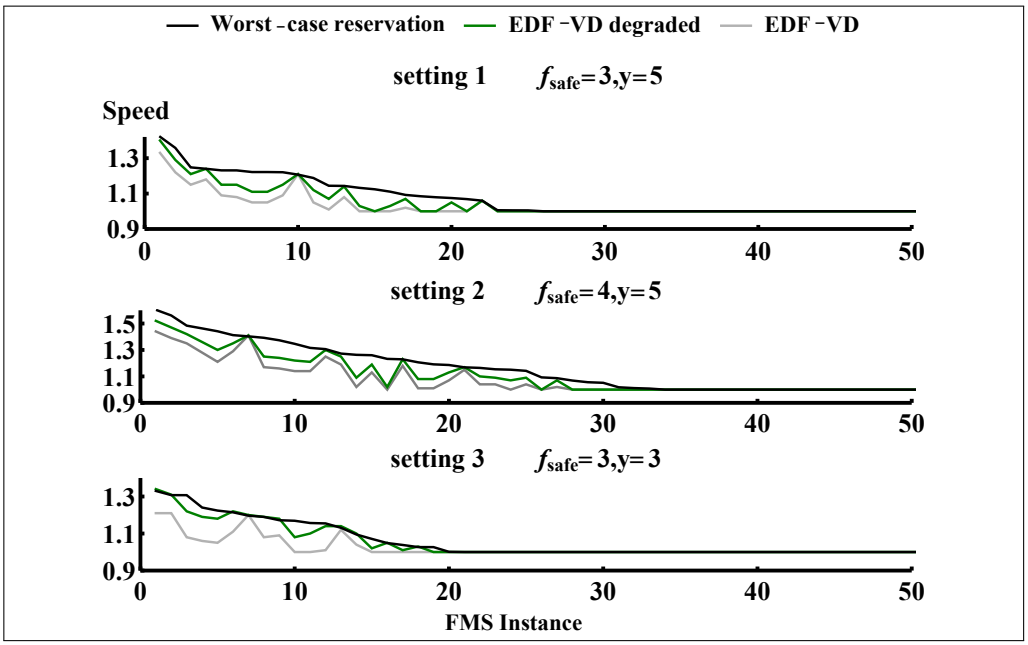


Figure 2.5: Processor speedup factors comparison – y-axis represents the processor speedup factor, x-axis represents FMS instances

the service resetting time can be bounded to certify the scheduling of FMS.

We consider a subset of the original FMS, which contains the localization and the flightplan tasks (DO-178B level B and level C, where B corresponds to HI criticality and C corresponds to LO criticality). All tasks are abstracted as implicit-deadline sporadic tasks. Typical ranges of WCETs are assumed. We show the task parameters in Table 2.2 with timing information in units of *ms*. A safety margin factor f_{safe} can be used to scale the WCETs on level C in order to get the WCETs on level B. Formally, we assume that $\forall \tau_i \in \tau_{\text{HI}}, C_i(\text{HI}) = f_{\text{safe}} * C_i(\text{LO})$. We generate a set of FMS instances with random level C WCETs conforming to Table 2.2.

We compare three different approaches: EDF with worst-case reservation (i.e., all tasks are guaranteed with WCETs on HI criticality level), EDF-VD with degraded service guarantee (i.e., a degraded service requirement for LO criticality tasks is guaranteed in HI criticality mode), and the original EDF-VD scheduling technique (all LO criticality tasks are rejected in HI criticality mode). In all three cases, we calculate the minimum speedup factors of the processor (by linear search) such that guarantees as aimed by those approaches are provided for FMS. A smaller speedup factor in this context means better schedulability.

We show in Figure 2.5 the speedup factors of all 3 approaches for 50 randomly generated FMS instances. The results are shown for 3 settings as indicated in Figure 2.5. Based on the experiment results, the original EDF-VD scheduling technique always has the minimum speedup factors

among all three approaches. This is intuitive since schedulability of HI criticality tasks is guaranteed by complete rejection of LO criticality tasks. However, EDF-VD is not applicable to FMS, as a degraded service requirement for LO criticality tasks must be provided. One solution to achieve this is by EDF scheduling with worst-case reservation. However, as one can expect, this approach always has the maximum processor speedup factors among all three approaches. The extension of EDF-VD with degraded service guarantee has a processor speedup factor in between of the other two approaches. The above observations are also confirmed by our theoretical analysis in condition (2.21): EDF with no service degradation corresponds to the case when $y = 1$, and EDF-VD with complete rejection of LO criticality tasks corresponds to the case when $y = \infty$; with y decreasing from ∞ to 1, the total system utilization will constantly increase by condition (2.21), leading to worse schedulability. Based on those results, we conclude that resource efficiency can be achieved by our proposed method in comparison to worst-case reservation, while degraded service for LO criticality tasks can be guaranteed in comparison to the original EDF-VD.

Furthermore, as shown in Figure 2.5, if we increase f_{safe} (setting 2), then the processor speedup factor will also increase (the maximum speedup factors of all three approaches in this case come close to 1.5, while the maximum speedup factors come close to 1.35 in setting 1). In addition, if we increase the degraded service for LO criticality tasks (i.e., decrease y in setting 3), the extension of EDF-VD with degraded service guarantee always has close or equal processor speedup in comparison to the worst-case reservation approach. This implies that the gain in resource saving for our extension of EDF-VD decreases with decreasing y.

We continue to quantify the service resetting time for the FMS usecase. For this purpose, we pick one randomly generated FMS instance. We evaluate the impact of f_{safe} and y on the resetting time. The results are shown in Table 2.3 with resetting times given in units of second. For $f_{\text{safe}} = 3$, according to Algorithm 2.1, both x and y equal to 1. The FMS needs not be reconfigured in this case as HI criticality WCETs of all tasks can be guaranteed. For $f_{\text{safe}} = 4$, the minimal y we calculate is 3, with a corresponding service resetting time of $21.6s$. If we increase y (i.e., decrease the degraded service for LO criticality tasks), we can reduce the resetting time ($y = 4, \Delta_R = 7.8s$). If we continue to increase f_{safe} to 5, the minimum y becomes 22 in this case, which has an associated service resetting time of $2.1 \times 10^3 s$.

2.2.5 Summary of Results

Table 2.3: Resetting time (Δ_R) for FMS in units of second

$f_{\text{safe}} = 3$			$f_{\text{safe}} = 4$			$f_{\text{safe}} = 5$		
y	x	Δ_R	y	x	Δ_R	y	x	Δ_R
1	1	0	3	0.25	21.6	22	0.25	2.1×10^3
-	-	-	4	0.25	7.8	23	0.25	661.8
-	-	-	5	0.25	5.92	24	0.25	406.1

Section 2.2 proposed the service adaptation model for mixed-criticality systems. Our proposed model includes the state-of-the-art model as a special case, and provides both explicit guarantees for LO criticality tasks in the urgent scenario and guarantees on when the system can resume to the nominal scenario. With EDF scheduling and an industrial flight management system, we demonstrated the feasibility of such a new model as well as various trade-offs.

2.3 The Interference-Constraint Graph Model

We proceed in this section by reducing the pessimism of the commonly assumed mixed-criticality model [BD16] from a different perspective. The rationale is that a designer would want to be able to specify and control explicitly system runtime behaviors – under what circumstances, which tasks in which applications may be affected by a task overrun. In the flight management system, there are different tasks associated with the active, secondary, and temporary flightplans. Tasks for the active flightplan are certified at level B (when A is the highest, and E is the lowest), while tasks for the secondary and temporary flightplan are certified at level C. When a high criticality task executes above a certain execution time threshold, the system may stop only tasks responsible for the secondary, or for the temporary, but not for both even if they are of the same criticality level. Otherwise, the pilot does not have any means to insert a new flightplan and to update the active one. A similar problem can occur if the run-time of a high criticality task in one application can lead to the removal of all low criticality tasks in other applications.

Most of the existing mixed-criticality scheduling strategies assume that they can stop a set of tasks based only on information about their criticality level without taking into account the actual functional requirements for these tasks. While this is commonly done in research

on schedulability, the required functionality of the safety-critical system may be compromised.

Currently mixed-criticality system designers do not have any means to specify which tasks can be affected if a certain task executes above a certain execution time threshold, and in what order the tasks can be affected. Moreover, stopping all tasks of a particular criticality level is often unnecessary as schedulability of the high criticality tasks can be guaranteed even when certain low criticality tasks continue executing. This is illustrated in the following example and confirmed by the experimental results in Section 2.3.3.

Example 2.3. *Consider a simple dual-criticality task set with three periodic tasks and deadlines smaller than or equal to periods. The parameters are given in Table 2.4 with time information in units of ms.*

Table 2.4: Task parameters for Example 2.3

	T	D	χ	C(LO)	C(HI)
τ_1	10	10	high	3	5
τ_2	6	6	low	3	3
τ_3	15	11	low	1	1

This task set is schedulable by existing scheduling techniques, e.g., EDF-VD [BF11]. However, it is not hard to observe that when τ_1 executes at C(HI), dropping only τ_3 is sufficient to guarantee the schedulability of τ_1. Therefore, the deadline of τ_2 can always be guaranteed under all runtime scenarios. □

This has been observed already in [PK11, SGTG12], and the authors propose scheduling mechanisms that improve the guarantees given to low criticality tasks by not unnecessarily stopping them. However, these approaches do not have any means to specify the order and scenarios in which tasks may interfere with each other. Here, we say a task interferes with another one if overrun of the former task leads to the removal of the latter task. A designer needs to be able to specify which task interferences are allowed by the functionality of the system and only they should be observable in any runtime scenario. Furthermore, a new scheduling algorithm is needed that is able to take a task interference specification and produce only schedules that meet this specification at runtime.

The motivation of this work is to propose a generalized specification of the allowed interferences in mixed-criticality task sets. The new specification overcomes the limitations described above. Moreover, it can easily be used to express a special case - the common practice that any high criticality task can interfere with all lower criticality tasks. Task sets specified with this new specification can be tested for schedulability with a fixed priority method which is presented and discussed. In particular,

our work makes the following contributions:

1 A new specification for mixed-criticality systems is proposed: The allowed interferences between tasks are specified as an Interference Constraint Graph (ICG). In this graph, each task maps to a node, and an edge quantifies the interference between a pair of tasks.

2 We then investigate how to design a schedule to satisfy a given ICG specification. We show that the Audsley's algorithm [AD91] can be used to design a fixed-priority scheduling policy which can meet a given ICG specification. This result resembles many previous results on mixed-criticality scheduling policies [Ves07, BBD11b]. It demonstrates that the more generic ICG specification is still susceptible to a simple and efficient scheduling policy.

3 Experimental results based on randomly generated task sets show that in many cases stopping all low criticality tasks is not necessary during runtime in order to guarantee schedulability of high criticality tasks. In particular, we show that interferences that tasks suffer can be reduced systematically, and an algorithm that removes edges from an ICG graph is proposed.

2.3.1 System Model

We assume again a set of n sporadic tasks denoted as $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$ executed on a uniprocessor. Each task τ_i is characterized by the following parameters: (1) A minimum inter-arrival time $T_i \in \mathbb{R}^+$ between any two successive jobs. (2) A relative deadline $D_i \in \mathbb{R}^+$ where $D_i \leq T_i$.

The actual execution time of a job of a task is not known until runtime when the job signals that it has finished executing. Based on this assumption, for the further discussion, we denote with $s(t)$ the runtime *scenario* at time t, $t \in \mathbb{R}^+$. A scenario is defined as the tuple $(c_1(t), c_2(t), \ldots, c_n(t))$, where $c_i(t)$ is the maximum of the run times of all jobs of task τ_i up to and including time t.

Each scenario $s(t)$ has at least one associated trace $tr(t)$ which leads to this scenario. The trace is characterized simply by the deadlines, arrival and finishing times of jobs up to and including time t.

Note that a scenario $s(t)$ and a corresponding trace $tr(t)$ depend on the arrival and execution times of jobs as well as on the selected scheduling strategy. In general, a task set and an associated scheduling policy may exhibit at runtime many scenarios and associated traces.

We restrict the scope of this work to consider only online scheduling strategies which cannot know the execution time of a job until running the job to completion.

2.3.1.1 Interference Constraint Graph

We propose the Interference Constraint Graph (ICG) as a specification of mixed-criticality task sets where the designer can specify the allowed interferences between tasks during runtime. We will discuss the syntax, semantics, and the properties of such a graph. We will present an example to illustrate the construction of such a graph specification from a commonly used mixed-criticality system model.

Definition 2.1. ***Interference Constraint Graph.*** *An Interference Constraint Graph (ICG) is a directed graph $G(V, E, \sigma)$, where V is a set of vertices which corresponds to the set of tasks, i.e., $V \stackrel{def}{=} \tau$, E is a set of directed edges with $e_{i,j} \in E$ being an edge from τ_i to τ_j, i.e., $e_{i,j} = (\tau_i, \tau_j)$, and $\sigma(e)$ is a function $\sigma : E \to \mathbb{R}^+$ defined for all edges $e \in E$.*

If there exists an edge $e_{i,j} = (\tau_i, \tau_j)$, then we say that task τ_i can *interfere with* task τ_j, meaning that if during runtime the execution time of a job of task τ_i exceeds $\sigma(e_{i,j})$ at time $t \geq 0$, then after time t, jobs of task τ_j do not need to be executed anymore. For all $e_{i,j} \in E$, we have $\sigma(\tau_i, \tau_j) \leq D_i$. Formally, we have the following schedulability definition.

Definition 2.2. ***ICG-Schedulability.*** *Given a task set τ and a corresponding ICG G, the task set is ICG-schedulable with a particular scheduling strategy if in any scenario $s(t)$ and any corresponding trace $tr(t)$ that can be produced by the task set and the strategy, jobs of any task τ_j with absolute deadlines smaller or equal to t must meet their deadlines if for no task τ_i with $e_{i,j} \in E$ we have that:*

$$c_i(t) > \sigma(\tau_i, \tau_j)\,.$$

In other words, ICG-Schedulability requires that deadlines of a task τ_j to be met in any scenario $s(t)$, only if each interfering task τ_i does not execute for more than $\sigma(\tau_i, \tau_j)$ in this scenario. If a task τ_j does not have any interfering tasks, i.e., there are no edges $e_{i,j}$, then jobs of task τ_j should meet their deadlines in any scenario $s(t)$.

Since we consider only online scheduling strategies, the above schedulability condition must be met for all t', $0 \leq t' \leq t$ in a scenario $s(t)$ and trace $tr(t)$.

Note that the ICG definition does not pose any restrictions on the topology of the graph. In particular, self-loops are quite useful as they can specify the maximum allowed execution time for a job of a task. The graph does not need to be complete, i.e., there may be no edges between some vertices. In particular, the graph may have no edges at all, meaning

that no tasks can interfere with each other. Checking the consistency of ICGs is outside of the scope of the current work.

2.3.1.2 ICG Representation of a Mixed-criticality Specification

Let us consider a mixed-criticality system model which is commonly used in literature [BV08, BLS10, LB10, LB12, BBD+12, EY12]. Each task in the system is associated with a single safety-criticality level assigned from the set $\chi \subseteq N^+$. The system may have an arbitrary number of criticality levels. In addition to the mentioned task parameters (periods and deadlines), we have that each task τ_i is characterized by the following parameters (slightly modified as compared to Section 2.1 for the convenience of this work):

- A safety-criticality level $\chi_i \in \chi$, where a bigger value denotes a higher safety-critical level.
- A set of worst-case execution time estimates $\hat{C}_i = \{C_i(\chi) \mid C_i(\chi) \in R^+, \chi \in \chi, C_i(\chi) = C_i(\chi_i) \forall \chi \geq \chi_i\}$. We assume that $C_i(\chi)$ is monotonically non-decreasing with increasing χ.

The criticality level of a scenario $s(t)$ is determined as the smallest value l ($l \in \chi$) that satisfies the inequality $c_i(t) \leq C_i(l)$ for all i, $1 \leq i \leq n$. We assume that l always exists, otherwise the scenario is considered erroneous.

Conventionally, a mixed-criticality task set is considered *schedulable* with a particular scheduling strategy if in any runtime scenario $s(t)$ with criticality level l and corresponding trace $tr(t)$ that the task set and the strategy can produce, the strategy will give sufficient execution to each job that belongs to a task with $\chi_i \geq l$ and has a deadline smaller or equal to t, such that the job can signal finishing before its deadline. This schedulability definition has been widely used in literature [BV08, BLS10, LB10, PK11, GESY11, LB12, BBD+12, EY12, SGTG12].

We can now demonstrate that the conventional schedulability condition can be easily expressed with an ICG. The following corollary follows from the definition of an ICG, and shows that it is possible to systematically construct an ICG for a mixed-criticality task set that is subject to the above schedulability condition.

Corollary 2.1. *A task set which is subject to the above system model and schedulability condition can be specified with an ICG graph $G = (V, E, \sigma)$ as follows: (1) $V = \tau$. (2) For every task, τ_i, add edge $e_{i,i} = (\tau_i, \tau_i)$, with $\sigma(e_{i,i}) = C_i(\chi_i)$. (3) For every pair of tasks τ_i and τ_j with $\chi_i > \chi_j$, add an edge $e_{i,j} = (\tau_i, \tau_j)$ with $\sigma(e_{i,j}) = C_i(\chi_j)$.*

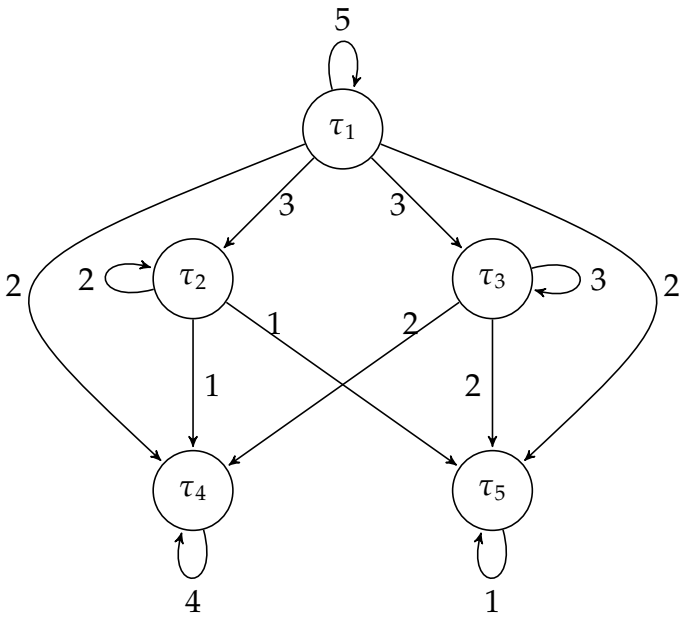


Figure 2.6: ICG representation for Example 2.4

The following example illustrates the results of the corollary and shows that the ICG specification is general enough to capture the commonly used mixed-criticality system model.

Example 2.4. *Consider a task set of five tasks with three different safety-criticality levels. The parameters of the tasks are given in Table 2.5.*

By Corollary 2.1, this commonly used task set specification can be represented with a corresponding ICG which is shown in Figure 2.6. Notice again that, the ICG specification is not limited to the standard specification, e.g., if τ_2 is not allowed to interfere with τ_4, it can be easily modeled by removing the edge between the two in the ICG. □

Table 2.5: Task parameters for Example 2.4

τ	T	D	χ	$C(3)$	$C(2)$	$C(1)$
τ_1	15	15	3	5	3	2
τ_2	10	10	2	2	2	1
τ_3	20	20	2	3	3	2
τ_4	30	30	1	4	4	4
τ_5	8	8	1	1	1	1

Most mixed-criticality scheduling policies implicitly specify which criticality levels are stopped under certain runtime scenarios. On the other hand, the ICG specification explicitly defines, between pairs of tasks, that a task may be dropped if a certain condition is satisfied. The current proposal considers only one type for the condition: exceeding

a certain execution time. However, it is not hard to see that this can be extended to include more general conditions such as deadline miss ratios, missing k deadlines in n consecutive instances, and others. On the other hand, one may not only consider dropping tasks, but also adjusting the Quality of Service provided to them, e.g., adjusting their periods or execution times at runtime.

2.3.1.3 Properties of ICG-Schedulability

Based on the ICG-Schedulability Definition 2.2, we can derive two useful properties with respect to the schedulability of an ICG.

The first one shows that whenever new interferences are introduced between tasks in an ICG-Schedulable task set, the ICG-Schedulability is maintained. Formally, this is specified as the Additivity property.

Property 2.1. ***Additivity.*** *Given a task set τ specified with ICG $G = (V, E, \sigma)$ which is ICG-Schedulable. If we add more edges to G, i.e., we obtain $G' = (V', E', \sigma')$, where $V' = V$, $E \subset E'$, and $\forall e \in E : \sigma'(e) = \sigma(e)$, ICG-Schedulability is maintained, i.e., G' is also ICG-Schedulable.*

Proof. Consider an ICG-Schedulable graph G where tasks τ_i and τ_j cannot interfere with each other, i.e., $(\tau_i, \tau_j) \notin E$. Let us consider a feasible trace tr where both tasks τ_i and τ_j execute, τ_i has finished before τ_j has arrived, and the execution time of τ_i is c_i.

Let us now consider the ICG G' with the added edge (τ_i, τ_j). Considering the value of $\sigma'(\tau_i, \tau_j)$, we have two scenarios. In the first one, suppose that $\sigma'(\tau_i, \tau_j)$ is greater or equal to c_i, then the execution trace is not modified. In the second scenario, suppose that $\sigma'(\tau_i, \tau_j)$ is less than c_i. In this scenario, the trace tr is modified by dropping instances of τ_j when the run time of τ_i exceeds $\sigma'(\tau_i, \tau_j)$, which does not change the feasibility of the trace. □

On the other hand, one may show that removing edges from an ICG may make the task set unschedulable. In reality, a designer may be interested in removing interferences between tasks while still maintaining the ICG-Schedulability property satisfied. A heuristic algorithm that removes the maximum number of edges from a given ICG while keeping schedulability satisfied is presented in Section 2.3.3.

The next property shows that increasing existing interferences between tasks of an ICG-Schedulable task set will keep ICG-Schedulability property satisfied. In particular, increasing the interference that a task τ_j can suffer from another task τ_i, i.e., decreasing the value of $\sigma(\tau_i, \tau_j)$, will keep ICG-Schedulability satisfied. Formally, this is specified as the

Monotonicity property.

Property 2.2. ***Monotonicity.*** *Given a task set τ specified with ICG $G = (V, E, \sigma)$ which is ICG-Schedulable. If we decrease the values of the function σ, i.e., we obtain $G' = (V', E', \sigma')$, where $V' = V$, $E' = E$, and $\forall e \in E : \sigma'(e) \leq \sigma(e)$, ICG-Schedulability is maintained, i.e., G' is also ICG-Schedulable.*

Proof. Consider an ICG-Schedulable graph G, where task τ_i can interfere with task τ_j with $\sigma(\tau_i, \tau_j)$. Let us consider a feasible trace tr where both tasks τ_i and τ_j execute, τ_i executes before τ_j, and the execution time of τ_i is c_i. Let us consider two cases. *Case 1:* $c_i > \sigma(\tau_i, \tau_j)$, in this case, when the run time of task τ_i exceeds $\sigma(\tau_i, \tau_j)$, task τ_j is dropped. Decreasing the value of $\sigma(\tau_i, \tau_j)$ will only cause task τ_j to be dropped earlier and the newly obtained trace is still feasible. *Case 2:* $c_i \leq \sigma(\tau_i, \tau_j)$, in this case, task τ_j is not dropped because of interferences from task τ_i. When decreasing $\sigma(\tau_i, \tau_j)$, we have two scenarios. Suppose that the new value is $\sigma'(\tau_i, \tau_j)$. If $\sigma'(\tau_i, \tau_j)$ is greater or equal to c_i, then the trace will not be modified. On the other hand, if $\sigma'(\tau_i, \tau_j)$ is less than c_i, then task τ_j will be dropped when the run time of τ_i exceeds $\sigma'(\tau_i, \tau_j)$, which does not change the feasibility of the trace. □

The two properties of additivity and monotonicity state the transformations of ICG graphs under which the system schedulability is preserved. We will show later in this section how we can use such properties "reversely" in order to minimize the interferences tasks suffer.

2.3.2 Fixed Priority Scheduling under ICG

The ICG specification as introduced in the previous section increases the expressiveness of existing mixed-criticality specifications by modeling explicitly the interferences between tasks. One intuition on the ICG semantics is that this more detailed specification would lead to increased complexity in the required scheduling strategy. In this section, we show that a fixed-priority preemptive scheduling strategy is applicable. Tasks are scheduled preemptively based on their statically assigned priorities.

Let us denote the set of higher priority tasks for a task τ_i as $hp(\tau_i)$. Further, define $\sigma(\tau_i, \tau_j) \stackrel{def}{=} +\infty$ when no directed edge from τ_i to τ_j exists in the ICG. We can adapt previous results on response time analysis for preemptive fixed priority scheduling [Ves07, BF11] and derive a schedulability test which takes into account interferences between tasks. An ICG-Schedulability test under fixed-priority preemptive scheduling is given in the following lemma.

Lemma 2.6. *Given a task set τ and a corresponding ICG $G = (V, E, \sigma)$, τ is ICG-Schedulable if $\forall \tau_i \in \tau$, we have that $R_i \leq D_i$, where R_i is a fixed-point solution of the following recurrence relation:*

$$R_i = \sigma(\tau_i, \tau_i) + \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot \min\left\{\sigma(\tau_j, \tau_j), \sigma(\tau_j, \tau_i)\right\}. \tag{2.27}$$

Proof. According to Definition 2.2, a task τ_i is guaranteed to meet its deadline only if each interfering task τ_j does not execute for more than $\sigma(\tau_j, \tau_i)$. We can consider two cases:

1. If a higher priority task τ_j can interfere with τ_i, it only needs to be assumed to execute for not more than $\min\{\sigma(\tau_j, \tau_j), \sigma(\tau_j, \tau_i)\}$ because:
   - If $\sigma(\tau_j, \tau_j) \geq \sigma(\tau_j, \tau_i)$, τ_i is dropped if the runtime of τ_j exceeds $\sigma(\tau_j, \tau_i)$.
   - If $\sigma(\tau_j, \tau_j) < \sigma(\tau_j, \tau_i)$, τ_j cannot trigger the dropping of τ_i, since it is not allowed to execute for more than $\sigma(\tau_j, \tau_j)$.
2. If a higher criticality task τ_j cannot interfere with τ_i, then its worst-case execution time is assumed to be $\sigma(\tau_j, \tau_j)$ when testing the schedulability of τ_i.

Both cases can be compactly represented as $\min\{\sigma(\tau_j, \tau_j), \sigma(\tau_j, \tau_i)\}$. □

Let us illustrate now the results of Lemma 2.6 with a simple example. First, let us denote that a task τ_i has higher priority than τ_j with $\tau_i \succ \tau_j$.

Example 2.5. *Consider a task set consisting of four tasks. Task parameters are given in Table 2.6 with $D_i = T_i, \forall i$. The corresponding ICG is shown in Figure 2.7.*

Table 2.6: Task parameters for Example 2.5

τ	τ_1	τ_2	τ_3	τ_4
T	15	22	12	6

In this example, task τ_1 is not allowed to interfere directly with τ_4. Hence, when considering the schedulability of τ_4, τ_1 must be assumed to take 6 units of execution time, if the schedulability of τ_4 depends on the execution time of τ_1 (e.g., τ_1 is assigned a higher priority than τ_4). In contrast, τ_1 is allowed to interfere with τ_2, hence when testing the schedulability of τ_2, τ_1 only needs to be assumed executing for 2 time units.

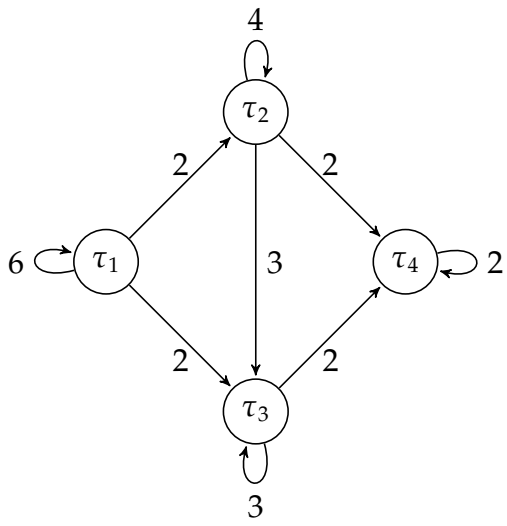


Figure 2.7: ICG for the task set in Example 2.5

Let us consider a particular priority ordering: $\tau_4 \succ \tau_1 \succ \tau_2 \succ \tau_3$. For the schedulability test of τ_3, following Lemma 2.6, the fixed-point of the following recursive equation gives the worst-case response time of τ_3:

$$R_3 = 3 + \left\lceil \frac{R_3}{15} \right\rceil \times 2 + \left\lceil \frac{R_3}{22} \right\rceil \times 3 + \left\lceil \frac{R_3}{6} \right\rceil \times 2. \tag{2.28}$$

Here, the execution times of τ_1 and τ_2 are assumed to be 2 and 3 units, since τ_3 does not need to be guaranteed when τ_1 and τ_2 execute more than 2 and 3 units, respectively. Task τ_4 is assumed to execute its worst-case execution time since it is not allowed to interfere with τ_3. Equation 2.28 has a fixed-point of 12, hence τ_3 is schedulable on the lowest priority level. In fact, this task set is ICG-Schedulable with the priority assignment given above. □

The schedulability test in Lemma 2.6 is sufficient but not necessary. The presented simple test does not take into account: 1) the interferences higher priority tasks suffer from low priority tasks, and 2) the interferences among the higher priority tasks. An exact schedulability test under fixed priority scheduling considering all interferences will need to combine all possible interference scenarios that can happen during runtime. However, this is outside of the scope of this work.

2.3.2.1 Priority Assignment

Following Lemma 2.6, the schedulability of a task is independent of the relative priority orderings of its higher priority tasks. Based on this, one can immediately test which tasks can be assigned on the lowest priority level using relation (2.27). Once the task on the lowest priority level is determined, it is removed from the task set. We continue to find the task

that can be assigned the second lowest priority level from the reduced task set, and so on. This process is recursively called until: 1) no task is left to be assigned a priority; or 2) at some step no task can be assigned on the current priority level (in this case the process fails to find a feasible schedule). This approach is known as the Audsley's algorithm [AD91].

It has been shown in [DB11] that Audsley's Optimal Priority Assignment algorithm is compatible with a schedulability test, if the following three conditions are satisfied:

1. The schedulability of a task is independent of the relative priority ordering of higher priority tasks.

2. The schedulability of a task is independent of the relative priority ordering of lower priority tasks.

3. If we swap the priorities of any two tasks of adjacent priority, then the task assigned higher priority cannot become unschedulable, if it was previously schedulable at the lower priority.

In particular, for a schedulability test complying with these three conditions, Audsley's algorithm is an optimal priority assignment approach, i.e., it will always find a feasible priority ordering as long as there exists one. This leads us to the following result on priority assignment for the ICG schedulability test presented in Lemma 2.6.

Theorem 2.6. *Audsley's algorithm [AD91] is an optimal priority assignment approach with respect to the schedulability test given in Lemma 2.6.*

Proof. We need to show that the test in Lemma 2.6 satisfies the three conditions described above.

1. The summation in relation (2.27) does not take into account the relative priority ordering of higher priority tasks.

2. Relation (2.27) does not consider any lower priority task because of the preemptive fixed priority strategy.

3. Consider two tasks τ_i and τ_j with priorities k and $k+1$, respectively. The upper bound on the response time of task τ_j cannot increase when it is assigned priority k, as the only change in the computation in relation (2.27) is the removal of task τ_i from the set of higher priority tasks.

□

2.3.3 ICG Evaluation

We demonstrate in this section the advantage of ICG specification: for mixed-criticality task sets with corresponding ICG specifications, we can achieve a systematic reduction of the interferences among tasks. We quantitatively measure the reduction of interferences by the percentage of edges that remain after edge removal in an ICG. For this purpose, an algorithm that removes edges from a given ICG is proposed, and numerical results on randomly generated task sets are presented.

2.3.3.1 Random Task Set Generation

The first step of our experiments is to randomly generate task sets with corresponding ICG graphs. This is done by first generating standard sporadic mixed-criticality task sets with implicit deadlines. The corresponding ICG specifications of the generated task sets are produced using the results of Corollary 2.1.

For generating the standard mixed-criticality task sets, we use a modified version of the random task generator as proposed in [LB12, BBD+12]. The random task generator is controlled by the following parameters:

- $[U_-, U_+]$: utilizations of tasks are uniformly drawn from this range, $0 < U_- < U_+ \leq 1$;
- U_{bound}: the system utilization bound, which is defined as $U_{\text{bound}} \stackrel{def}{=} \max_{\chi \in \boldsymbol{\chi}} \{ \sum_{\{\tau_i | \chi_i \geq \chi\}} \frac{C_i(\chi)}{T_i} \}$;
- r: the ratio of worst-case execution times of any task at any two consecutive criticality levels ($\frac{C(\chi+1)}{C(\chi)}$);
- $[R_-, R_+]$: r is uniformly drawn from this range, $1 \leq R_- < R_+$;
- $[T_-, T_+]$: the periods of tasks are uniformly drawn from this range;
- P_χ: the probability that any task is of criticality χ; $\sum_{\chi \in \boldsymbol{\chi}} P_\chi = 1$.

2.3.3.2 Edge Removing

As shown by Property 2.1, adding new edges in the ICG specification will maintain ICG-Schedulability. On the other hand, removing edges while the task set stays schedulable can be used to reduce the interferences tasks suffer. We use the percentage of remaining edges in an ICG after edge removal as a quantitative measure of the improved expressiveness of the proposed specification.

Algorithm 2.2: Interference Minimization

Input: τ, $G(V, E, \sigma)$
Output: $G'(V, E', \sigma')$
for $e \in E$ **do**
  calculate the minimum speedup factor s_e of the processor with only e in G, such that the task set is ICG-Schedulable; set the weight of edge e (contribution to schedulability, different from $\sigma(e)$): $w_e \leftarrow \frac{1}{s_e}$;
end
sort edges E in decreasing order of their weights;
$E' \leftarrow \emptyset$;
while τ *is not ICG-Schedulable under* $G'(V, E', \sigma')$ **do**
  add the first edge e in E to E' and remove e from E, $\sigma'(e) \leftarrow \sigma(e)$;
end
return $G'(V, E', \sigma')$;

The problem then is to maximally remove edges from a given ICG. We view the optimality criteria simply as removal of the maximum number of edges in the ICG specification while ensuring schedulability. In order to solve this problem, one intuition is that different edges in an ICG have different contributions to schedulability. Hence, we can first evaluate their weights in terms of such contributions. Once this is done, we can add edges in decreasing weights until ICG-Schedulability is satisfied. The proposed heuristic algorithm is shown in Algorithm 2.2.

Specifically, for each edge we measure the minimal speedup factor of the processor such that the task set is ICG-schedulable if only this edge exists in the ICG. Clearly, a lower speedup factor means a higher contribution to schedulability. Hence, the weight of an edge can be measured by the inverse of the corresponding speedup factor. The algorithm then tries greedily to add the minimum number of edges in the ICG by adding one edge at a time in decreasing order of their weights until the task set becomes schedulable. For calculation of the speedup factor under fixed priority scheduling in Algorithm 2.2, we use an existing technique as proposed in [DRRG10].

2.3.3.3 Results

We now apply Algorithm 2.2 to randomly generated mixed-criticality task sets, and we are interested in observing how many edges we can remove from the ICGs while maintaining schedulability. The experiments are extensively conducted on 1000 random task sets generated with specific group of controlling parameters.

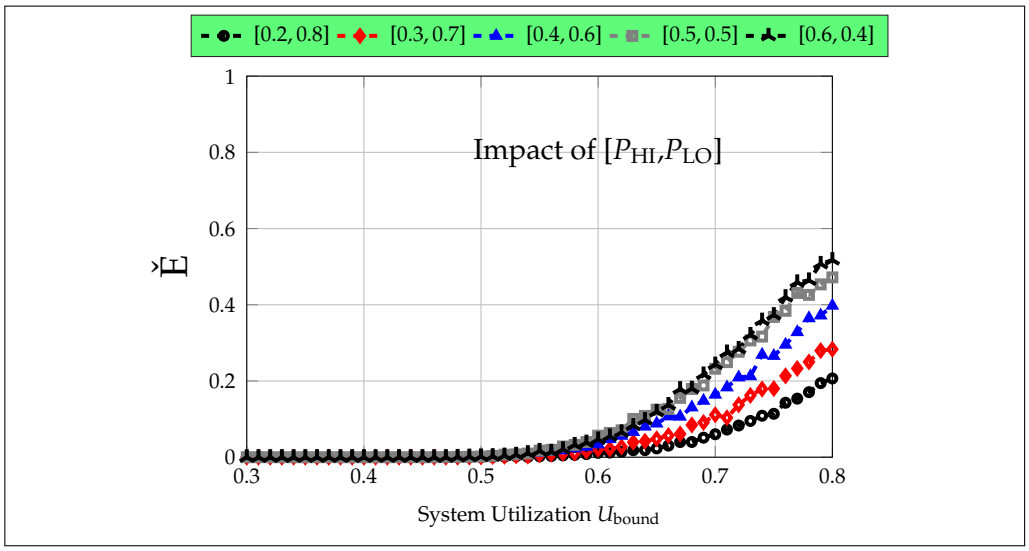


Figure 2.8: Ě with $T_- = 4, T_+ = 16, U_- = 0.02, U_+ = 0.2, R_- = 1, R_+ = 4$

Let us denote the set of schedulable task sets without any edge removal as Sched, then the average percentage of edges (Ě) *remaining* after edge removal for a set of task sets can be defined as follows:

$$\check{E} = \sum_{\tau \in \text{Sched}} \frac{|E'_\tau|}{|E_\tau|} \Big/ |\text{Sched}| \,. \tag{2.29}$$

Effect of ratio of high criticality tasks: Here we consider randomly generated dual-criticality task sets (HI for high criticality and LO for low criticality). In particular, we evaluate how the criticality-probability distribution (P_χ) will affect Ě. As shown in Figure 2.8, for the typical case when the high criticality tasks only constitute a small portion of the dual-criticality task set, considerable number of edges in the ICG graphs can be removed. For instance, when $P_{HI} = 0.2$, we may be able to keep at most around 20% of the edges in all tested cases in order to ensure schedulability. Intuitively if we increase P_{HI}, Ě would increase since more interferences with the low criticality tasks are needed to accommodate the extra workloads of the high criticality tasks.

Effect of ratio of worst-case execution times: Here we show how the ratio of worst-case execution times between two consecutive criticality levels will affect Ě. The experiments are conducted on random dual-criticality task sets where $P_{HI} = 0.2$. Intuitively, for smaller worst-case execution time ratios between consecutive criticality levels, the interferences low criticality tasks suffer should be less: for the extreme case when the ratio is equal to 1, interference with low criticality tasks will not improve schedulability and all edges in the ICGs can be removed. As shown by Figure 2.9, for task sets generated with ratios in a wider range,

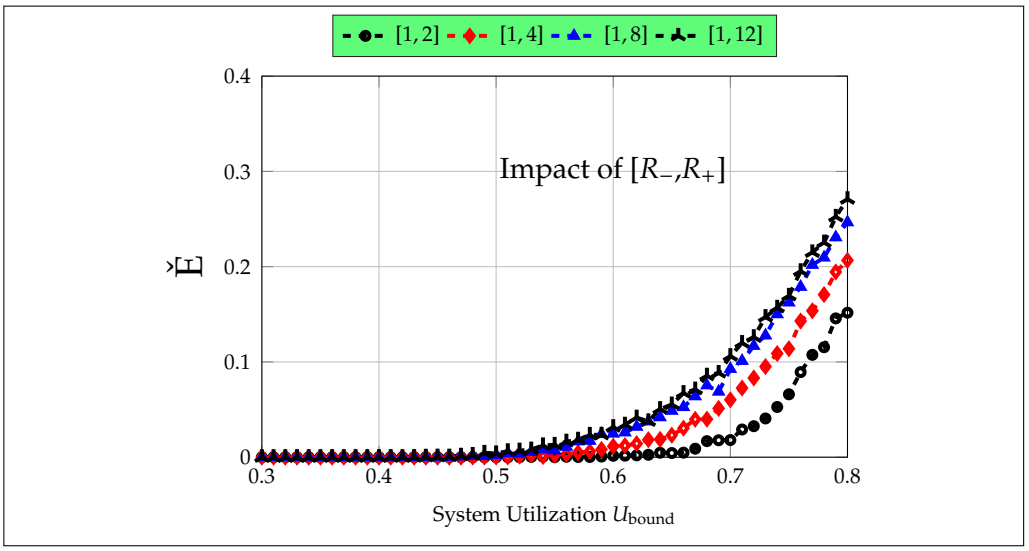


Figure 2.9: Ě with $T_- = 4, T_+ = 16, U_- = 0.02, U_+ = 0.2, P_{\mathrm{HI}} = 0.2, P_{\mathrm{LO}} = 0.8$

the percentages of remaining edges after edge removal are increased.

Effect of number of criticality levels: Here we present how the number of criticality levels will affect Ě. The larger the number of criticality levels in the system, the more interferences tasks can suffer. This is because the degree of uncertainties in worst-case execution times is higher. We evaluate the impact of the number of criticality levels in our experiments by generating ICG graphs for random task sets with 2 to 5 criticality levels, with equal criticality probabilities. As shown in Figure 2.10, the percentage of edges we have to keep in the ICG graphs increases with increasing number of criticality levels. Furthermore, we can observe, for higher number of criticality levels, tasks need to interfere with each other even when the system utilization bound is low, e.g., for 4 criticality levels, interferences between tasks are needed to ensure schedulability already when $U_{\mathrm{bound}} = 0.35$.

2.3.4 Summary of Results

Section 2.3 proposed the Interference Constraint Graph (ICG) model, and demonstrated its dominance over the standard mixed-criticality model regarding expressiveness; several interesting properties of this new model were elaborated. Our experimental results explored the structure of ICG to optimize (i.e., remove) interferences among tasks; as indicated by the test results, interferences in mixed-criticality systems can be reduced to a great extent under ICG.

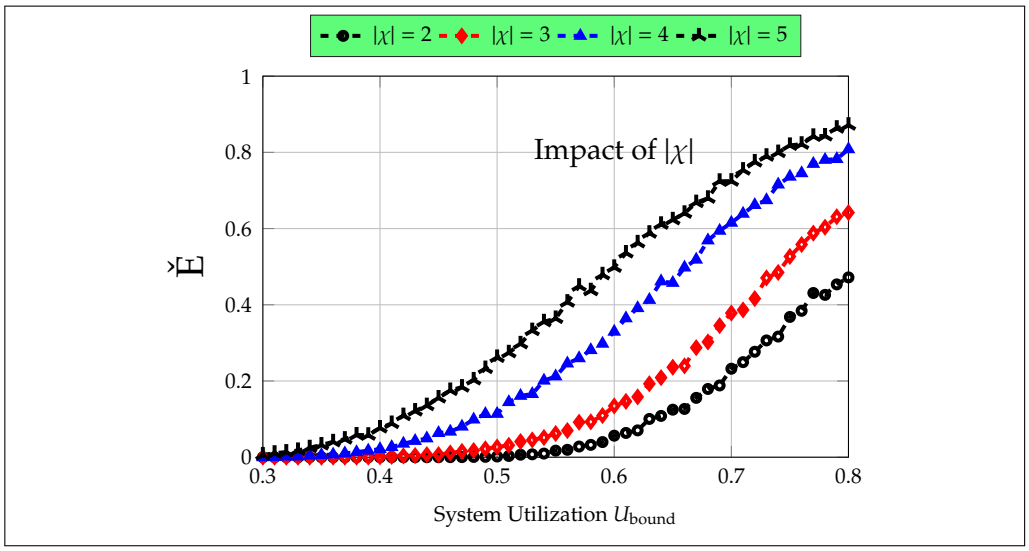


Figure 2.10: Ě with $T_- = 4, T_+ = 16, U_- = 0.02, U_+ = 0.2, R_- = 1, R_+ = 4$

2.4 Run and Be Safe: Mixed-Criticality Scheduling with Temporary Processor Speedup

We have addressed in Section 2.2 and Section 2.3 the limit of the classic mixed-criticality model from the software perspective (i.e., mixed-criticality scheduling protocols); in this section, we will address the same problem by exploring hardware features of modern computing platforms.

Our key insight is that, routine features of such platforms like dynamic voltage and frequency scaling (DVFS) [BBPDM99], and dynamic cache partitioning and locking (DCPL) [PA06] could facilitate the design of adaptive mixed-criticality systems. As a proof of concept, we solve in this section mixed-criticality scheduling with the aid of DVFS. DVFS is conventionally used to conserve energy by exploring free slacks in the system to slow the processor down (underclocking) [BBPDM99]. In contrast, we adopt DVFS to speed up the processor (overclocking) when there is a timing urgency caused by task overrun, such that all tasks could still meet their deadlines. If service degradation for less critical tasks is permitted under overrun, then speeding up the processor can improve the system degraded services.

In practice, processor speedup increases energy dissipation and is often regulated by power and thermal management [NRAW11]. For example, Intel turbo boost technology would allow a maximum of 2x speedup for around 30s [NRAW11]. Moreover, fast recovery from adversary events like task overrun is often desired, especially for embedded systems deployed in safety-related domains. In both regards, we show that processor speedup can actually help the system

to recover faster: By speeding up the processor, the system overload could be resolved faster, enabling quick resuming of normal operation. Nevertheless, in an extreme case when tasks overrun frequently and the system needs to overclock for a long period, then we could monitor overclocking at runtime – whenever it exceeds an allowed time budget, we could terminate tasks and reset the system to normal speed.

In this work, we assume the same mixed-criticality models in Section 2.2.1 and adopt earliest deadline first (EDF) scheduling. We provide theoretical results on: (1) calculating offline a minimum processor speedup to guarantee system schedulability in critical operation mode (Section 2.4.1), and (2) quantifying offline the system resetting time under processor speedup (Section 2.4.2). We analytically show the trade-offs between different design parameters including processor speedup, service degradation and resetting time in Section 2.4.3. To demonstrate the applicability of our proposed techniques, we conduct experiments with both an industrial flight management system and synthesized task sets (Section 2.4.4). Our results confirm that dynamic processor speedup can greatly increase system schedulability regions and is typically required for only short periods of time.

2.4.1 Minimum Processor Speedup to Handle Overrun

We present in this section how to compute the minimum required processor speedup to handle task overrun and guarantee task deadlines. For this purpose, we will first revisit some results on system schedulability analysis.

Since EDF is assumed in this work, we could use demand bound analysis to determine the system schedulability. The demand bound function for task τ_i in a time interval of length Δ is defined as the worst-case total execution times of all τ_i's jobs with both arrival times and deadlines in this time interval. Known results exist in case that task parameters are constant, e.g., in LO mode, the demand bound function of any task can be calculated as [EY13]:

$$\mathrm{DBF}_{\mathrm{LO}}(\tau_i, \Delta) = \max\left\{\left\lfloor \frac{\Delta - D_i(\mathrm{LO})}{T_i(\mathrm{LO})} + 1 \right\rfloor, 0\right\} \times C_i(\mathrm{LO}). \tag{2.30}$$

To guarantee *schedulability in* LO *mode*, it then suffices to show that the sum of demand bound functions for all tasks is no greater than the provided processing resource in any time interval [EY13].

The schedulability analysis in HI mode is non-trivial, since there could exist schedulability dependencies between LO and HI modes. This is because unfinished jobs in LO mode are forced to take the processing resources in HI mode to meet their HI mode deadlines. This effect needs

to be taken into account when calculating the demand bound functions in HI mode. We adopt in this work a known result to calculate such HI mode demand bound functions [EY13, HGST14]:

Lemma 2.7. *For any task $\tau_i \in \tau$, define a set of functions as follows:*

$$w(\tau_i, \Delta) = (\Delta \bmod T_i(\text{HI})) - (D_i(\text{HI}) - D_i(\text{LO})), \tag{2.31}$$

$$r(\tau_i, \Delta, w(\cdot)) = \begin{cases} \min\{w(\tau_i, \Delta), C_i(\text{LO})\} & \text{if } w(\tau_i, \Delta) \geq 0 \\ \quad +C_i(\text{HI}) - C_i(\text{LO}), & \\ 0, & \text{otherwise} \end{cases}. \tag{2.32}$$

The demand bound function of τ_i in HI *mode can be calculated as:*

$$DBF_{\text{HI}}(\tau_i, \Delta) = r(\tau_i, \Delta, w(\cdot)) + \left\lfloor \frac{\Delta}{T_i(\text{HI})} \right\rfloor \cdot C_i(\text{HI}). \tag{2.33}$$

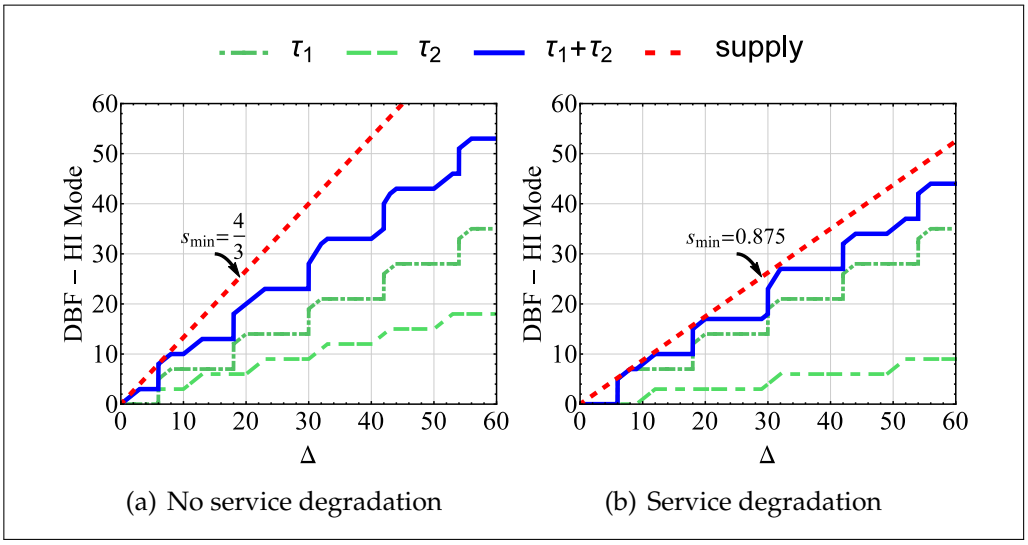
Notice that in Lemma 2.7, $r(\tau_i, \Delta, w(\cdot))$ essentially quantifies the impact of unfinished jobs in LO mode on HI mode system demands. Furthermore, we see that HI mode system demands depend on: (1) the preparation for task overrun in LO mode, i.e., LO mode deadlines of HI criticality tasks, and (2) the degraded services in HI mode, i.e., new deadlines and job inter-arrival times for LO criticality tasks.

Based on the above calculations of demand bound functions, we can now compute a minimum processor speedup to *guarantee schedulability in* HI *mode*.

Theorem 2.7. *Assume the processor speeds up by a factor s when entering* HI *mode to handle overrun. The minimum processor speedup factor, $s_{\min}$, which guarantees* HI *mode schedulability, can be calculated as follows:*

$$s_{\min} = \max_{\Delta \geq 0} \left\{ \sum_{\tau_i \in \tau} DBF_{\text{HI}}(\tau_i, \Delta) \Big/ \Delta \right\}. \tag{2.34}$$

Proof. Based on the demand bound analysis, to guarantee HI mode schedulability, it is sufficient [EY13, HGST14] that $\sum_{\tau_i \in \tau} \text{DBF}_{\text{HI}}(\tau_i, \Delta) \leq s \cdot \Delta, \forall \Delta \geq 0$. We can then reformulate and derive $s \geq \max \left\{ \sum_{\tau_i \in \tau} \text{DBF}_{\text{HI}}(\tau_i, \Delta) \Big/ \Delta \right\}, \forall \Delta \geq 0$. Thus, we have equation (2.34). □



(a) No service degradation (b) Service degradation

Figure 2.11: Minimum speedup and demand bound functions

Notice that in equation (2.34), we allow the divisor Δ to be zero: If the total HI mode system demands are non-zero in an interval of zero length, the system will require infinite speedup in HI mode, i.e., $s_{\min} = +\infty$. This could actually happen if the deadlines of HI criticality tasks are not shortened in LO mode, see our discussions regarding relation (2.2).

Table 2.7: Example task set

τ	χ	C_i(LO)	C_i(HI)	D_i(LO)	D_i(HI)	T_i(LO)	T_i(HI)
τ_1	HI	2	7	4	10	12	12
τ_2	LO	3	3	6	6	10	10

Example 2.6. *Consider a simple dual-criticality task set with parameters shown in Table 2.7. The* LO *criticality task τ_2 here needs to adhere to its original service parameters in* HI *mode. With dynamic processor speedup allowed, we can apply equation (2.34) and calculate the minimum required speedup for* HI *mode is $\frac{4}{3}$. Furthermore, if we allow τ_2 to degrade its service in* HI *mode by setting $D_2(\text{HI}) = 15, T_2(\text{HI}) = 20$, then the required speedup factor can be reduced to 0.875. In this case, the system can actually slow down in* HI *mode despite the fact that τ_1 overruns, since the system load is greatly reduced by degrading the services for τ_2. We plot in Figure 2.11 our results for both cases. As we can see, the computed minimum speedup factors do guarantee* HI *mode schedulability.*

Computation efficiency: Equation (2.34) suggests that the computation of the minimum processor speedup would require examination of all non-negative interval lengths. However, in practice, this computation can be done efficiently in pseudo-polynomial time. First, due to the

periodicity of the demand bound function (2.33), we can limit the maximum examined interval length to the least common multiple of all HI mode task minimum inter-arrival times. Second, only a list of discrete interval lengths needs to be checked: As suggested by equation (2.33), the demand bound function is piecewise-linear, which can be also observed in Figure 2.11. Therefore, only boundaries at those piecewise-linear segments need to be checked. Formally, we have the following results.

Lemma 2.8. *Denote the least common multiple of all tasks' HI mode minimum inter-arrival times as LCM_T, and define a set of functions as follows:*

$$pd(\tau_i, k) = D_i(\text{HI}) - D_i(\text{LO}) + C_i(\text{LO}) + kT_i(\text{HI}), \tag{2.35}$$

$$\hat{bd}(\tau_i, l) = \{pd(\tau_i, k) \mid k \in \mathbb{N} \wedge pd(\tau_i, k) \leq l\}, \tag{2.36}$$

$$\check{bd}(\tau_i, l) = \{pd(\tau_i, k) - C_i(\text{LO}) \mid k \in \mathbb{N} \wedge pd(\tau_i, k) - C_i(\text{LO}) \leq l\}. \tag{2.37}$$

The minimum processor speedup $s_{\min}$, which guarantees HI mode schedulability, can be computed in pseudo-polynomial time:

$$s_{\min} = \max_{\Delta \in \left(\bigcup_{\tau_i} \hat{bd}(\tau_i, LCM_T)\right) \cup \left(\bigcup_{\tau_i} \check{bd}(\tau_i, LCM_T)\right)} \left\{ \sum_{\tau_i \in \tau} DBF_{\text{HI}}(\tau_i, \Delta) \Big/ \Delta \right\}. \tag{2.38}$$

Proof. We prove in two steps. **First**, we will show that the maximum interval length to be examined can be limited to the least common multiple of all tasks' HI mode minimum inter-arrival times (LCM_T). According to equation (2.33), the HI mode demand bound functions have periodic patterns, i.e.,

$$\begin{aligned} \text{DBF}_{\text{HI}}(\tau_i, \delta + kT_i(\text{HI})) &= \text{DBF}_{\text{HI}}(\tau_i, kT_i(\text{HI})) + \text{DBF}_{\text{HI}}(\tau_i, \delta) \\ &= k \cdot \text{DBF}_{\text{HI}}(\tau_i, T_i(\text{HI})) + \text{DBF}_{\text{HI}}(\tau_i, \delta) \\ &= kC_i(\text{HI}) + \text{DBF}_{\text{HI}}(\tau_i, \delta), \ \forall k \in \mathbb{N}. \end{aligned} \tag{2.39}$$

Thus, $\forall\ 0 \leq \delta \leq \text{LCM}_T,\ \forall k \in \mathbb{N}$, we have:

$$\begin{aligned} &\frac{\sum_{\tau_i \in \tau} \text{DBF}_{\text{HI}}(\tau_i, \delta + k\text{LCM}_T)}{\delta + k\text{LCM}_T} \\ =\ &\frac{\sum_{\tau_i \in \tau} \text{DBF}_{\text{HI}}(\tau_i, \delta) + \sum_{\tau_i \in \tau} \text{DBF}_{\text{HI}}(\tau_i, k\text{LCM}_T)}{\delta + k\text{LCM}_T} \\ =\ &\frac{\sum_{\tau_i \in \tau} \text{DBF}_{\text{HI}}(\tau_i, \delta) + k \cdot \sum_{\tau_i \in \tau} \text{DBF}_{\text{HI}}(\tau_i, \text{LCM}_T)}{\delta + k\text{LCM}_T}. \end{aligned} \tag{2.40}$$

Now, $\forall\ 0 \leq \delta \leq \text{LCM}_T \wedge \frac{\sum_{\tau_i \in \tau} \text{DBF}_{\text{HI}}(\tau_i, \delta)}{\delta} \geq \frac{\sum_{\tau_i \in \tau} \text{DBF}_{\text{HI}}(\tau_i, \text{LCM}_T)}{\text{LCM}_T}$, we have:

$$\frac{\sum_{\tau_i \in \tau} \text{DBF}_{\text{HI}}(\tau_i, \delta) + k \cdot \sum_{\tau_i \in \tau} \text{DBF}_{\text{HI}}(\tau_i, \text{LCM}_T)}{\delta + k\text{LCM}_T} \leq \frac{\sum_{\tau_i \in \tau} \text{DBF}_{\text{HI}}(\tau_i, \delta)}{\delta}. \tag{2.41}$$

Notice that, such δ always exists (e.g., $\delta = \text{LCM}_T$), and equation (2.40) will increase as long as $\frac{\sum_{\tau_i \in \tau} \text{DBF}_{\text{HI}}(\tau_i, \delta)}{\delta}$ increases. Hence, we have:

$$\max_{\Delta \leq \text{LCM}_T} \left\{ \sum_{\tau_i \in \tau} \text{DBF}_{\text{HI}}(\tau_i, \Delta) \middle/ \Delta \right\} \geq \max_{\Delta > \text{LCM}_T} \left\{ \sum_{\tau_i \in \tau} \text{DBF}_{\text{HI}}(\tau_i, \Delta) \middle/ \Delta \right\}. \tag{2.42}$$

Second, we prove that only a list of discrete interval lengths needs to be examined. According to equation (2.33), the HI mode demand bound function of any task will increase in only intervals $[D_i(\text{HI}) - D_i(\text{LO}) + kT_i(\text{HI}), D_i(\text{HI}) - D_i(\text{LO}) + C_i(\text{LO}) + kT_i(\text{HI})]$, where $k \in \mathbb{N}$. Furthermore, we observe that, for such intervals, the HI mode demand bound function is piecewise linear for all tasks due to equation (2.33) (as can be also seen in Figure 2.11). Let us denote one such segment of $\sum_{\tau_i \in \tau} \text{DBF}_{\text{HI}}(\tau_i, \Delta)$ as $\alpha \cdot \Delta + \beta$. Clearly, the maximum value of $\frac{\sum_{\tau_i \in \tau} \text{DBF}_{\text{HI}}(\tau_i, \Delta)}{\Delta}$ (i.e., $\alpha + \beta/\Delta$) in this segment, can only be achieved at the boundaries of it. By considering all such segments and their boundaries, we have equation (2.38). □

Runtime: During runtime, whenever any HI criticality task exceeds its LO criticality WCET, the system transits to HI mode and the processor is speeded up by $s_{\min}$. The offline bound of $s_{\min}$ ensures that all tasks will meet their deadlines whenever the system enters HI mode.

2.4.2 Service Resetting Time under Processor Speedup

We proceed to show that speeding up the processor to handle task overrun is even more attractive, as it can help the system to recover normal operation faster. Together with the fact that task overrun is rare, this suggests processor speedup can incur low cost as it would only be temporarily required.

To quantify service resetting time under processor speedup, we need to first identify a sufficient condition under which a system can be safely reset to LO mode. Since we do not know statically how long the system will overrun, we can safely assume that the system can recover when the processor is idle, i.e., when all arrived demands are finished. Here, we say that the demand (in terms of WCET) of a job "arrives" whenever the job arrives. Notice that, the system may idle many times in HI mode

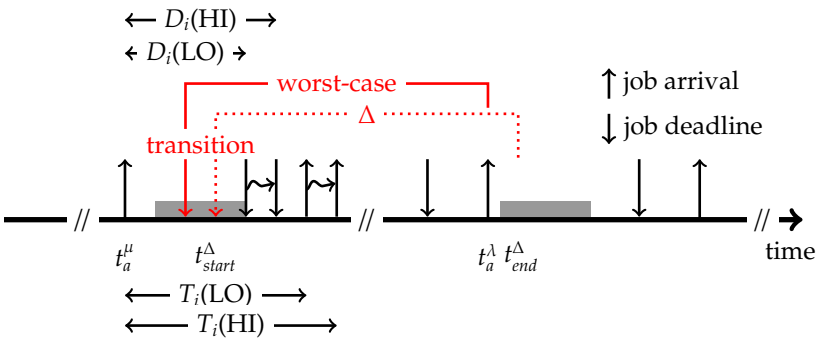


Figure 2.12: Worst-case scenario for arrived demand

whenever the arrived jobs are finished; therefore, we need to find the closest idle time after entering HI mode, at which point the system can safely recover.

As a second step, we need to bound the worst-case arrived system demands *starting* from the transition to HI mode, as this is required to upper-bound when all arrived demands can be firstly finished in HI mode. Special attention needs to be paid to unfinished jobs in LO mode, since those "partial" jobs can be viewed as if they are released at the time of mode transition and must finish in HI mode. In fact, there is already one known result [HGST14] in the literature which solves this problem. However, the technique proposed therein relies essentially on exhaustive search to find the worst-case arrived demands after entering HI mode. In the following, we will analytically identify such worst-case scenario.

For presentation convenience, we assume the system transits to HI mode at time $\hat{t}$. In addition, let us denote the start and end of a time interval of length Δ as t_{start}^Δ and t_{end}^Δ, respectively. Furthermore, we denote the *latest* jobs of τ_i arriving no later than t_{start}^Δ and t_{end}^Δ as μ and λ, respectively. The arrival time of these two jobs are denoted as t_a^μ, t_a^λ, respectively. Figure 2.12 depicts graphically our notations.

Lemma 2.9. *For a time interval of length Δ starting from the transition to* HI *mode ($t_{start}^\Delta = \hat{t}$), the worst-case arrived demand for any task $\tau_i \in \tau$ is obtained when this time interval ends with a future job arrival of τ_i, i.e., $t_{end}^\Delta = t_a^\lambda$.*

Proof. Let us assume $t_{end}^\Delta > t_a^\lambda$. Now assume we shift the considered time interval to the left such that the new end time of the time interval t'^Δ_{end} equals to t_a^λ. In this case, the demand of job λ still arrives within the time interval. Furthermore, for job μ, we distinguish between two cases:

1 $t_{start}^\Delta \leq t_a^\mu + D_i$(LO): In this case, job μ could be unfinished at transition as it is only expected to finish by $t_a^\mu + D_i$(LO) in LO mode. By shifting

the considered interval to the left, job μ can only have executed less in LO mode, and more remaining load is "carried into" HI mode.

2 $t^{\Delta}_{start} > t^{\mu}_{a} + D_i(\text{LO})$: Here, job μ has already finished when the system transits to HI mode. By shifting the time interval interested to the left, job μ could become unfinished if the time interval now starts before or at $t^{\mu}_{a} + D_i(\text{LO})$, i.e., it starts to "carry" load into HI mode.

Since the jobs arriving in between μ and λ are not affected, and the carried-in demand of μ is non-decreasing, it follows that by setting the considered time interval ending with one job arrival of τ_i, the arrived demands of τ_i in the time interval cannot decrease. This concludes the proof. □

The essential idea to prove Lemma 2.9 is to show that, by letting $t^{\Delta}_{end} = t^{\lambda}_{a}$ (shift the dotted time interval in Figure 2.12 to the left), the released demands of all jobs including μ and λ are not decreased. Now, based on Lemma 2.9, we can quantify the worst-case arrived demand for any task starting from the transition to HI mode and bound when the system can be safely recovered.

Theorem 2.8. *Assume the processor speeds up by a factor s when entering* HI *mode at time $\hat{t}$, and define function $w'(\cdot)$ as follows:*

$$w'(\tau_i, \Delta) = (\Delta \bmod T_i(\text{HI})) - (T_i(\text{HI}) - D_i(\text{LO})). \tag{2.43}$$

The worst-case arrived demand bound of τ_i in time interval $[\hat{t}, \hat{t} + \Delta]$ can be calculated as:

$$ADB_{\text{HI}}(\tau_i, \Delta) = r(\tau_i, \Delta, w'(\cdot)) + \left(\left\lfloor \frac{\Delta}{T_i(\text{HI})} \right\rfloor + 1\right) \cdot C_i(\text{HI}), \tag{2.44}$$

where function $r(\cdot)$ is defined in equation (2.32).

If the system is idle at time $\hat{t} + \Delta_R$, it must follow that:

$$\sum_{\tau_i \in \tau} ADB_{\text{HI}}(\tau_i, \Delta_R) \leq s \cdot \Delta_R. \tag{2.45}$$

Proof. According to Lemma 2.9, for any task τ_i, the worst-case arrived demand happens for interval $[\hat{t}, \hat{t} + \Delta]$ if it ends with a job arrival of τ_i. In this case, the maximum number of arrived jobs after transiting to HI mode is bounded by $\left\lfloor \frac{\Delta}{T_i(\text{HI})} \right\rfloor + 1$.

In addition, we have to consider the "carried-in" demand of the unfinished job of τ_i. We can calculate that, in the worst-case, the transition to HI mode happens $w'(\tau_i, \Delta)$ before job μ's deadline in LO mode. If

$w'(\tau_i, \Delta) < 0$, then μ is finished in LO mode and the carried-in demand is zero. Otherwise, the demand of this unfinished job is increased by $C_i(\text{HI}) - C_i(\text{LO})$ due to mode transition. Moreover, the remaining LO mode execution of μ can be calculated as $\min\{w'(\tau_i, \Delta), C_i(\text{LO})\}$. This is because μ is only expected to finish $C_i(\text{LO})$ units of execution by its LO mode deadline before the mode transition and the unfinished execution of job μ is at most $C_i(\text{LO})$. It then follows that the carried-in demand of job μ can be compactly represented by $r(\tau_i, \Delta, w')$, with function $r(\cdot)$ defined in equation (2.32).

Finally, if the system is idle at time $\hat{t} + \Delta_R$, the total arrived demands of all tasks starting from entering HI mode must have been finished, which leads to condition (2.45). □

With the calculation of worst-case arrived demands starting from $\hat{t}$ in equation (2.44), the system is idle whenever the arrived demands are no greater than the supplied resources, as stated in condition (2.45). Since the system can potentially have many idling points in HI mode, we can then choose the first idling point after mode transition. Subsequently, we can calculate the time difference between this idling point and the time when the system transits to HI mode, which gives a safe lower-bound on the service resetting time. Formally, this can be formulated as follows.

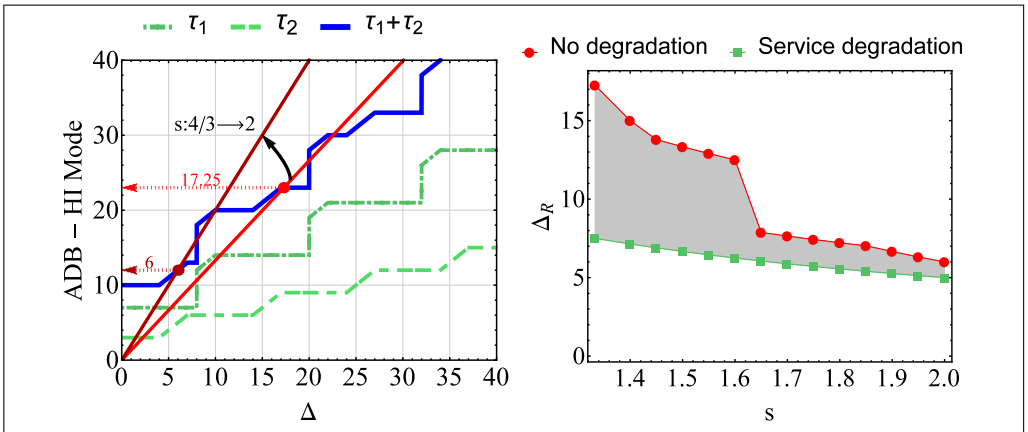
Corollary 2.2. *If the processor speed is increased by a speedup factor s after transiting to* HI *mode, a safe service resetting time can then be lower-bounded by*

$$\Delta_R = \min\left\{\Delta \geq 0 : \sum_{\tau_i \in \tau} ADB_{\text{HI}}(\tau_i, \Delta) \leq s \cdot \Delta\right\}. \tag{2.46}$$

Example 2.7. *Consider the same task set as shown in Table 2.7. According to equation (2.46), we can calculate the service resetting time for this task set. Our results are depicted in Figure 2.13.*

If no service degradation is allowed, we calculate that the service resetting time is 17.25 when s equals $\frac{4}{3}$. In addition, if s is increased to 2, then the service resetting time can be reduced to 6 since now overload is resolved faster with higher processor speed. This is shown in Figure 2.13(a).

The trend of service resetting time as s increases is better illustrated with Figure 2.13(b). As shown by the results, there is a clear gain if the dynamic processor speedup is increased. We will present how to derive closed-formulas to express this relation in Section 2.4.3. Furthermore, if service degradation is enabled in parallel to processor speedup, the service resetting time can be further reduced as less overload needs to be addressed. For our results here, the degraded service parameters as shown in Example 2.6 are adopted.



(a) No service degradation (b) Parametric trend

Figure 2.13: Service resetting time under dynamic processor speedup

Computation efficiency: Similar to the computation of the minimum processor speedup, calculation (2.46) can also be reduced to pseudo-polynomial complexity. Formally, we have the following result.

Lemma 2.10. *Define a set of functions as follows:*

$$pa(\tau_i, k) = T_i(\text{HI}) - D_i(\text{LO}) + C_i(\text{LO}) + kT_i(\text{HI}), \tag{2.47}$$

$$\hat{ba}(\tau_i, l) = \{pa(\tau_i, k) \mid k \in \mathbb{N} \wedge pa(\tau_i, k) \leq l\}, \tag{2.48}$$

$$\check{ba}(\tau_i, l) = \{pa(\tau_i, k) - C_i(\text{LO}) \mid k \in \mathbb{N} \wedge pa(\tau_i, k) - C_i(\text{LO}) \leq l\}, \tag{2.49}$$

$$A_{UB}(\Delta) = \sum_{\tau_i \in \tau} \left(C_i(\text{HI}) + \left(\max \left\{ \frac{C_i(\text{HI}) - C_i(\text{LO})}{T_i(\text{HI}) - D_i(\text{LO})}, \frac{C_i(\text{HI})}{T_i(\text{HI}) - D_i(\text{LO}) + C_i(\text{LO})} \right\} \right) \cdot \Delta \right). \tag{2.50}$$

Denote the cross point of $A_{UB}(\Delta)$ and $S \cdot \Delta$ in interval domain as $\Delta_{A_{UB}}$. The service resetting time can then be calculated in pseudo-polynomial time: For any two points pa_1 and pa_2 ($\{pa_1, pa_2\} \subset \left(\bigcup_{\tau_i} \hat{ba}(\tau_i, \Delta_{A_{UB}}) \right) \cup \left(\bigcup_{\tau_i} \check{ba}(\tau_i, \Delta_{A_{UB}}) \right)$), if

$$\sum_{\tau_i \in \tau} ADB_{\text{HI}}(\tau_i, pa_1) \geq s \cdot pa_1 \wedge \sum_{\tau_i \in \tau} ADB_{\text{HI}}(\tau_i, pa_2) \leq s \cdot pa_2, \tag{2.51}$$

then the system service resetting time exists within $[pa_1, pa_2]$.

Proof. According to equation (2.44), $\forall 0 \leq \delta \leq T_i(\text{HI}),\ \forall k \in \mathbb{N}$,

$$\text{ADB}_{\text{HI}}(\tau_i, \delta + kT_i(\text{HI})) = \text{ADB}_{\text{HI}}(\tau_i, \delta) + kC_i(\text{HI}). \tag{2.52}$$

Furthermore, based on equation (2.44), the arrived demand function for task τ_i only changes at zero interval length (increased to $C_i(\text{HI})$) and intervals $[T_i(\text{HI}) - D_i(\text{LO}) + kT_i(\text{HI}), T_i(\text{HI}) - D_i(\text{LO}) + C_i(\text{LO}) + kT_i(\text{HI})]$, where $k \in \mathbb{N}$. As a result, we can first upper-bound the slope of the arrived demand function:

$$\begin{aligned}
&\max_{0\le\delta\le T_i(\text{HI})\wedge k\in\mathbb{N}} \frac{\text{ADB}_{\text{HI}}(\tau_i, \delta + kT_i(\text{HI})) - C_i(\text{HI})}{\delta + kT_i(\text{HI})} \\
&= \max_{0\le\delta\le T_i(\text{HI})\wedge k\in\mathbb{N}} \frac{\text{ADB}_{\text{HI}}(\tau_i, \delta) - C_i(\text{HI}) + kC_i(\text{HI})}{\delta + kT_i(\text{HI})} \\
&\le \max_{0\le\delta\le T_i(\text{HI})} \frac{\text{ADB}_{\text{HI}}(\tau_i, \delta) - C_i(\text{HI})}{\delta} \text{ (similar to relations (2.40) and (2.41))} \\
&\le \max\left\{\frac{C_i(\text{HI}) - C_i(\text{LO})}{T_i(\text{HI}) - D_i(\text{LO})}, \frac{C_i(\text{HI})}{T_i(\text{HI}) - D_i(\text{LO}) + C_i(\text{LO})}\right\}.
\end{aligned} \tag{2.53}$$

Consequently, the total arrived demands of all tasks can be bounded by equation (2.50), the cross point of which and $s \cdot \Delta$ guarantees that for any point after this (larger interval length), the supplied processing resource is larger than the arrived demand. Thus, we get an upper-bound of the interval length to be checked. Furthermore, due to the piecewise linearity of the demand bound function, we can locate the minimum service resetting time as shown by relation (2.51) – one only needs to scan all the boundary points of linear segments to find such service resetting time, leading to pseudo-polynomial time complexity. □

In the above proof, we first upper-bound the total arrived demand bounds by a linear function. By comparing this linear upper-bound and the service supply ($s \cdot \Delta$), we can determine the maximum interval length to check (the cross point of the two). Second, the total arrived demands are piecewise linear w.r.t. Δ, e.g., see Figure 2.13(a). Thus, we only need to check the boundaries of each segment to determine whether the service supply crosses with the segment.

Runtime: During runtime, whenever the processor is idle, the system switches back to LO mode, and the processor speed is restored to its original speed. Our offline bound of resetting time predicts when such restoration can happen.

Remark: Our calculation of the service resetting time Δ_R does not assume any particular task overrun pattern. If we assume in the worst-case two bursts of overrun are separated by at least T_O units of time ($T_O \gg 0$ since overrunning expected WCET is rare), then the frequency that the system needs to speedup is bounded by $\frac{1}{T_O}$ as long as $\Delta_R \le T_O$. In typical cases, the time used to speedup (Δ_R) is often limited to a few tens of seconds due to power/thermal management, which would anyway satisfy this constraint if $\Delta_R \ll T_O$.

2.4.3 Special Case and System Level Trade-offs

To theoretically show the relation between different system parameters, the current research on mixed-criticality often adopts implicit-deadline tasks with two assumptions [BBD+12, HGST14]: (1) Deadlines of HI criticality tasks in LO mode are shortened by a common factor $0 < x < 1$ to prepare for overrun:

$$D_i(\text{LO}) = x \cdot D_i(\text{HI}), T_i(\text{HI}) = T_i(\text{LO}) = D_i(\text{HI}), \forall \tau_i \in \tau_{\text{HI}}. \tag{2.54}$$

(2) Deadlines of LO criticality tasks are scaled up by a common factor $y \geq 1$ in HI mode to react to overrun:

$$D_i(\text{HI}) = y \cdot D_i(\text{LO}), T_i(\chi) = D_i(\chi), \forall \tau_i \in \tau_{\text{LO}}\ \forall \chi. \tag{2.55}$$

We continue to show that with those assumptions, closed-formula solutions can be derived for computing the minimum required processor speedup and the service resetting time. In the following, we will again denote $\frac{C_i(\chi)}{T_i(\chi)}$ as $U_i(\chi)$, where $\chi \in \{\text{HI}, \text{LO}\}$.

2.4.3.1 Processor Speedup

We first present the following result on bounding how overrun preparation x and service degradation y will affect the required speedup.

Lemma 2.11. *Given design parameters x and y, the minimum required speedup to guarantee* HI *mode schedulability can be upper-bounded as:*

$$s_{\min} = \sum_{\tau_{\text{HI}}} \max \left\{ \frac{U_i(\text{HI})}{U_i(\text{LO})+(1-x)}, \frac{U_i(\text{HI})-U_i(\text{LO})}{1-x} \right\} + \sum_{\tau_{\text{LO}}} \frac{U_i(\text{LO})}{U_i(\text{LO}) + (y-1)}. \tag{2.56}$$

Proof. According to Lemma 2.7, the demand bound function of any task τ_i in HI mode only increases linearly in intervals $[D_i(\text{HI}) - D_i(\text{LO}) + kT_i(\text{HI}), D_i(\text{HI}) - D_i(\text{LO}) + C_i(\text{LO}) + kT_i(\text{HI})]$, where $k \in \mathbb{N}$. Thus, we only need to check for the boundaries of such intervals for the maximum value of $\text{DBF}_{\text{HI}}(\tau_i, \Delta)/\Delta$. Furthermore, due to the periodicity of the demand bound function, we can limit such boundaries to when $k = 0$ (similar to the bound of arrived demand function as shown in Lemma 2.10). Based on equation (2.33), at time $D_i(\text{HI}) - D_i(\text{LO})$ the demand of τ_i is increased by $C_i(\text{HI}) - C_i(\text{LO})$, and at $D_i(\text{HI}) - D_i(\text{LO}) + C_i(\text{LO})$ the task's demand is further increased by $C_i(\text{LO})$. The maximum value of $\text{DBF}_{\text{HI}}(\tau_i, \Delta)/\Delta$ at the two points gives the corresponding upper-bound:

$$\text{DBF}_{\text{HI}}(\tau_i, \Delta) \leq \left(\max \left\{ \frac{C_i(\text{HI})-C_i(\text{LO})}{D_i(\text{HI})-D_i(\text{LO})}, \frac{C_i(\text{HI})}{D_i(\text{HI})-D_i(\text{LO})+C_i(\text{LO})} \right\} \right) \cdot \Delta. \tag{2.57}$$

Taking the assumptions in this section ($T_i(\text{HI}) = D_i(\text{HI}), \forall \tau_i$), we have:

$$\begin{aligned}\sum_{\tau_i \in \tau} \text{DBF}_{\text{HI}}(\tau_i, \Delta) &\leq \sum_{\tau_i \in \tau} \left(\max \left\{ \frac{C_i(\text{HI}) - C_i(\text{LO})}{T_i(\text{HI}) - D_i(\text{LO})}, \frac{C_i(\text{HI})}{T_i(\text{HI}) - D_i(\text{LO}) + C_i(\text{LO})} \right\} \right) \cdot \Delta, \\ \sum_{\tau_i \in \tau} \text{DBF}_{\text{HI}}(\tau_i, \Delta) \Big/ \Delta &\leq \sum_{\tau_i \in \tau} \left(\max \left\{ \frac{C_i(\text{HI}) - C_i(\text{LO})}{T_i(\text{HI}) - D_i(\text{LO})}, \frac{C_i(\text{HI})}{T_i(\text{HI}) - D_i(\text{LO}) + C_i(\text{LO})} \right\} \right) \\ &= s_{\min} \text{ (see equation (2.34))}.\end{aligned} \tag{2.58}$$

Finally, with relations (2.54) and (2.55), calculation (2.58) can be simplified and equation (2.56) can be derived. □

Lemma 2.11 clarifies that $s_{\min}$ will monotonically decrease with decreasing x and/or increasing y. We explain the intuitions behind this along with the following example.

Example 2.8. *Consider the task set shown in Table 2.7, where task parameters are now modified according to relations (2.54) and (2.55). Based on Lemma 2.11, we depict in Figure 2.14(a) the impact of overrun preparation x and service degradation y on the minimum required speedup to guarantee* HI *mode schedulability. As we can see, for smaller x (i.e., more overrun preparation in* LO *mode), the minimum required speedup is decreased: By making* HI *criticality tasks finish earlier in* LO *mode, more resources are statically reserved for overrun in* HI *mode. Thus, less speedup is required to address overload. In addition, with higher service degradation (larger y), the required speedup also decreases. The reason is the same as explained for x: Degrading the service for* LO *criticality tasks more will reduce* HI *mode system load, leading to less required speedup.*

2.4.3.2 Service Resetting Time

We proceed to present how the service resetting time can be affected by system design parameters using a closed formula.

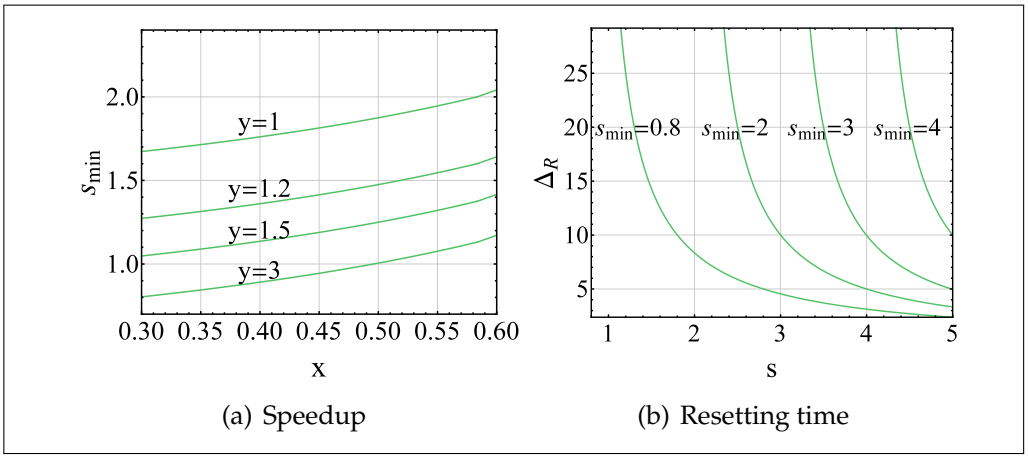
Lemma 2.12. *Given design parameters x, y and speedup s in* HI *mode, the service resetting time can be bounded as:*

$$\Delta_R = \frac{\sum_{\tau_i \in \tau} C_i(\text{HI})}{s - s_{\min}}, \tag{2.59}$$

where $s_{\min}$ is given by equation (2.56).

Proof. As shown in Lemma 2.10, we can bound the total arrived demand bound function as:

$$A_{UB}(\Delta) = \sum_{\tau_i \in \tau} \left(C_i(\text{HI}) + \left(\max \left\{ \frac{C_i(\text{HI}) - C_i(\text{LO})}{T_i(\text{HI}) - D_i(\text{LO})}, \frac{C_i(\text{HI})}{T_i(\text{HI}) - D_i(\text{LO}) + C_i(\text{LO})} \right\} \right) \cdot \Delta \right). \tag{2.60}$$



(a) Speedup (b) Resetting time

Figure 2.14: Various tradeoffs: Impact of overrun preparation x and service degradation y on required speedup and service resetting time

Taking relation (2.58), we get:

$$A_{UB}(\Delta) = \left(\sum_{\tau_i \in \tau} C_i(\text{HI}) \right) + s_{\min} \cdot \Delta. \tag{2.61}$$

Finally, with Corollary 2.2, equation (2.59) can be derived.

□

With Lemma 2.12, we can see there is a clear gain in service resetting time when processor speedup in HI mode is increased. In addition, the service resetting time would be $+\infty$ if the minimum processor speedup $s_{\min}$ is chosen.

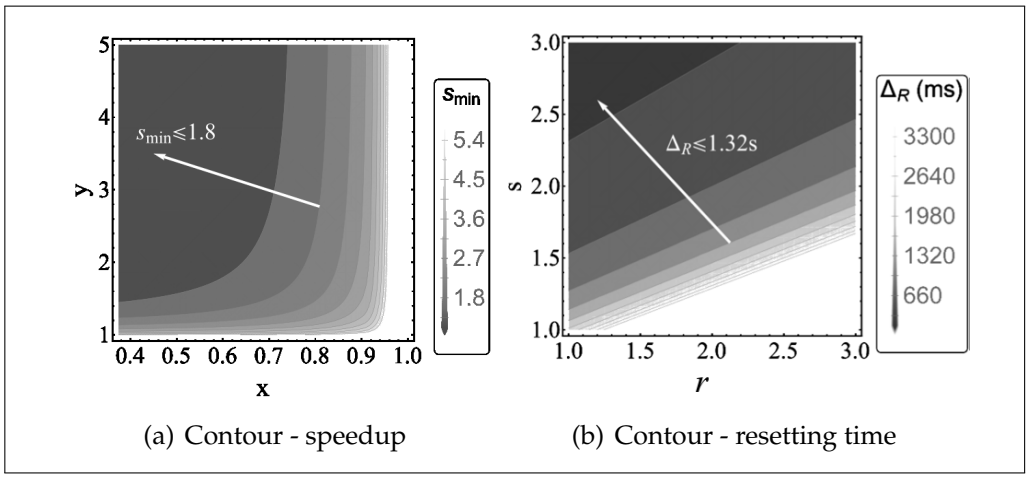
Example 2.9. *With the same task set used in previous examples, we plot in Figure 2.14(b) Δ_R against s with different minimum required system speedups. Essentially, $s_{\min}$ represents the system load in* HI *criticality mode. We observe, with artificially increased $s_{\min}$ (more system load in* HI *mode), the service resetting time is increased as it will take longer to resolve overload. Likewise, similar trend can be observed if we decrease the actual system speedup s.*

2.4.4 Evaluation

We present in this section the validation of our proposed techniques with an industrial flight management system (FMS) and synthetic task sets.

2.4.5 Flight Management System

We adopt a subset of an industrial implementation of FMS, which consists of 7 DO-178B criticality level B (HI) and 4 criticality level C (LO) tasks.



(a) Contour - speedup (b) Contour - resetting time

Figure 2.15: Experimental results on FMS

All tasks can be modeled as implicit deadline sporadic tasks, with task minimum inter-arrival times in the range of 100ms to 5s. For detailed parameters, we refer to Table 2.2. We present our results in Figure 2.15.

As suggested by Figure 2.15(a), with decreasing x (better safety preparation) or increasing y (more service degradation), the required speedup in HI mode to meet deadlines is reduced, see relations (2.54) and (2.55) for definitions of x and y. This is because that the system load in HI mode is decreased in both cases. The contour plot here clearly depicts the freedom in setting x and y for a certain speedup in HI mode. Furthermore, Figure 2.15(b) presents how the service resetting time is affected by the speedup in HI mode s and the uncertainty in HI criticality tasks' workloads r ($r := \frac{C_i(\text{HI})}{C_i(\text{LO})}, \forall \tau_i \in \tau_{\text{HI}}$). Our results show that with increasing r or decreasing s, the service resetting time is increased, as increased overload is resolved more slowly. In addition, we observe that FMS takes in the worst-case less than 3s to recover with a speedup of 2, indicating that dynamic processor speedup could indeed only be temporarily required.

2.4.6 Synthetic Task Sets

To show the applicability of our proposed techniques to general task sets, we now conduct extensive experiments on synthesized task sets. We adopt the random task generator as shown in Section 2.3.3.1. The task generator starts with an empty task set and continuously adds new random tasks to this set until certain system utilization U_{bound} is met. In the following, we generate 500 task sets at each system utilization point and conduct our experiments. Our results are shown in Figure 2.16, and we observe:

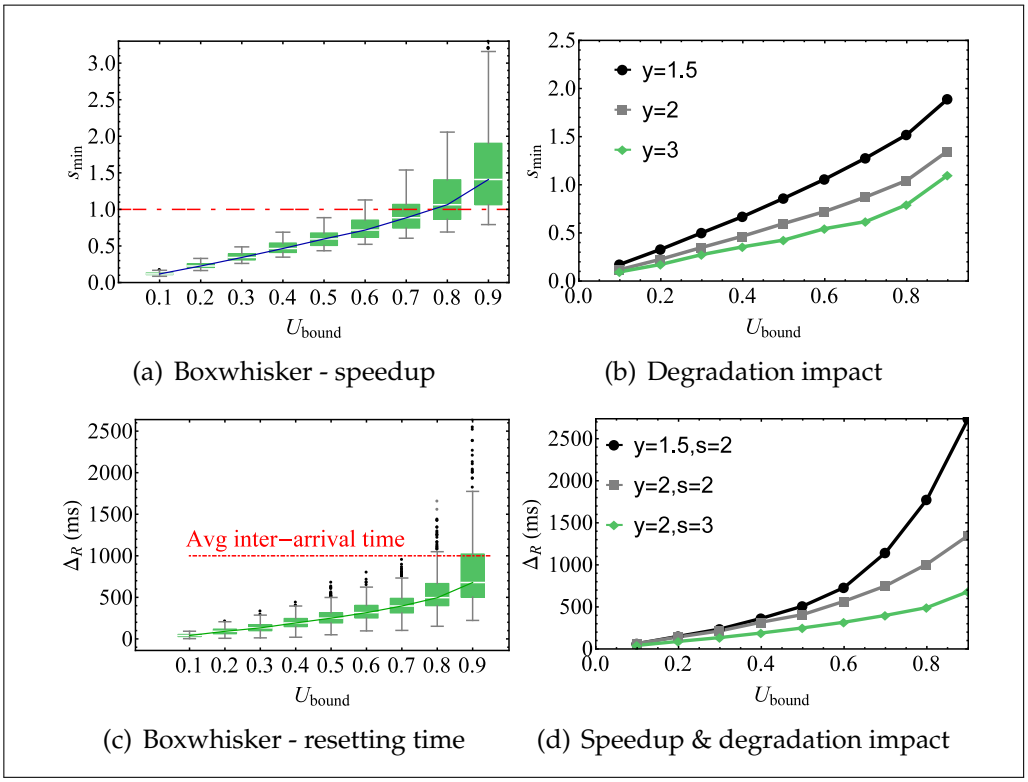


Figure 2.16: Experiments using synthesized task sets, with task minimum inter-arrival times randomly chosen from 2ms to 2s, task LO criticality utilization $C_i(\text{LO})/T_i(\text{LO})$ randomly chosen from 0.01 to 0.2 and $r = C_i(\text{HI})/C_i(\text{LO})$ ($\forall\tau_i \in \tau_{\text{HI}}$) randomly chosen from 1 to 3. Figure 2.16(a) is obtained assuming $y = 2$ and shows the distribution of s_{min}. Figure 2.16(c) is obtained assuming $y = 2, s = 3$ and shows the distribution of Δ_R. Figure 2.16(b) and Figure 2.16(d) show the median values of our results across different system utilizations. x in all cases is set to the minimum to guarantee LO mode schedulability [HGST14].

- Our proposed techniques can successfully bound the required processor speedup and the service resetting time in all tested cases. As the system utilization U_{bound} increases, both the required speedup and the service resetting time increase since more overload needs to be addressed in HI mode.

- The maximum required processor speedup is less than 3.3 for all tested cases ($U_{\text{bound}} = 0.9$ in Figure 2.16(a)). In this case, the 50th percentile value of processor speedup is only 1.4. In addition, for all cases when $U_{\text{bound}} \leq 0.5$, the maximum required speedup is less than 1, indicating that the system can even slow down in HI mode. Furthermore, we see that speedup indeed greatly improves system schedulability: Less than 25% task sets are schedulable when $U_{\text{bound}} = 0.9, s_{\text{min}} = 1$, which is increased to 75% when $s_{\text{min}} = 1.9$.

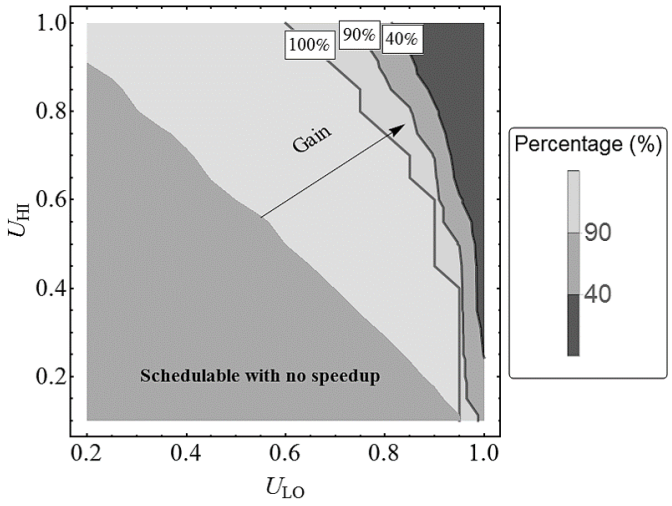


Figure 2.17: Schedulability region under temporary processor speedup: 2x speedup for no longer than 5s. For this set of experiments, r (defined in Figure 2.16) is set to 10. All other task generation parameters are the same as in Figure 2.16. $U_\chi = \sum_{\chi_i \geq \chi} C_i(\chi)/T_i(\chi), \chi \in \{\mathrm{HI}, \mathrm{LO}\}$. Marked numbers represent percentages of schedulable task sets.

- The longest service resetting time for all tested cases is less than 2.6s ($U_{\mathrm{bound}} = 0.9$ in Figure 2.16(c)), while the median value of resetting time in this case is only 678.6ms. Furthermore, we observe that the median/average service resetting time increases slowly with increased system utilization. However, the worst-case (dotted outliers in Figure 2.16(c)) can increase dramatically. In most test cases, the system can be reset within the average inter-arrival time of all tasks.

- With more service degradation, both the minimum required processor speedup and the system service resetting time can be reduced (increasing y for Figure 2.16(b) and Figure 2.16(d)), as less overload needs to be resolved in HI mode. Furthermore, with increasing processor speedup s in HI mode, the service resetting time can be further reduced, as convinced by the results in all test cases in Figure 2.16(d).

2.4.7 Schedulability Region

We finally present in Figure 2.17 the schedulability regions under dynamic processor speedup. For the experimental results, we set $s = 2$ and explicitly require that the resetting time must fulfill $\Delta_R \leq 5s$. We show how the system schedulability region could be increased with such temporary

processor speedup. For this purpose, we generate in total 6.84×10^4 random task sets at all $(U_{\text{HI}}, U_{\text{LO}})$ points and conduct our experiments, see Figure 2.17 for explanation of U_χ. We assume that LO criticality tasks are terminated in HI mode. At each $(U_{\text{HI}}, U_{\text{LO}})$ point, we consider a small neighboring region ($U_\chi \pm 0.025$), and compute the percentage the system is schedulable as the ratio of the number of schedulable task sets to the number of all task sets in this neighboring region.

Based on our results, we observe the following: First, the region in which the system is 100% schedulable is greatly increased compared to the case when no processor speedup is used. Furthermore, at high system utilization points, temporary processor speedup can still guarantee a large portion of schedulable task sets. For example, when both U_{HI} and U_{LO} equal 0.85, 90% task sets generated here can still be successfully scheduled with 2x processor speedup for no longer than 5s.

2.4.8 Summary of Results

Section 2.4 provided theoretical results as well as empirical evaluations, both demonstrating that over-clocking can help mixed-criticality systems not only to improve the degraded services for less critical tasks, but also to recover faster from the critical scenario.

2.5 A New Isolation Scheduling Model for Mixed-Criticality Systems

Conventionally, embedded system industries favor strict isolation among applications of different criticality levels due to ease of certification. Following the end of Dennard scaling [EBSA+11], embedded processors increasingly feature a multi-core architecture with shared resources (e.g., last-level cache, memory controller) in order to keep improving performance and efficiency (Figure 2.18 depicts an example of such an architecture). This, however, is challenging the best practice of criticality isolation due to contention in accessing shared resources:

- Coarse-grain static partitioning in time and space, e.g., based on the DO-178B standard [do11] for avionics or the ISO 26262 standard [iso11] for automotive systems, is an established technique for single-core safety-critical systems, but this approach can not be simply applied to multi-core architectures; it would only allow one job to execute at any point in time if strictly applied.

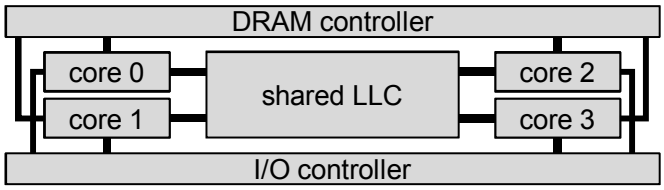


Figure 2.18: Sample typical multi-core architecture with $m = 4$ cores that share last-level cache, DRAM controller and I/O controller.

- More fine-grained partitioning requires to individually control the access to each shared resource [SCM+14].
- Finally, the approach of finding a global schedule and bounding the contention on shared resources at any time is only feasible with the knowledge of the detailed resource sharing behavior of all tasks, and it quickly becomes computationally intractable with an increasing number of tasks [GLST12].

In this work, we propose a new scheduling model that we call Isolation Scheduling (IS). IS is a practical model for efficiently scheduling real-time tasks on multi-core processors, i.e., exploiting hardware parallelism and shared resources. To make the problem more tractable, IS makes an assumption about the tasks, i.e., we assume that tasks are partitioned into *task classes* that have exclusive access to the processor and the platform resources. This way, interference on shared resources is greatly reduced; inter-class interference is eliminated by construction, and only intra-class interference needs to be considered. Well-established methods [GLST12, WKP13, GSH+15] can be applied to bound / control this remaining interference.

Indeed, subdividing real-time tasks into classes provides key benefits in several contexts. For instance, *gang scheduling* [ZFMS03, KI09, GB10] groups jobs (threads) that share information through fine-grained synchronization in the same class in order to reduce blocking times. Conversely, when jobs do not share information and there are well-defined task dependencies, communication takes place between classes in order to respect task dependencies, safely bound blocking times, and avoid concurrent access to shared memory. Furthermore, *server*-based scheduling [AB98] is an established approach for performance isolation among task classes with different timing requirements, e.g., for co-scheduling hard and soft real-time tasks or periodic and aperiodic tasks. Finally, in the context of *mixed-criticality systems* [Ves07], tasks are grouped in classes of different safety criticality, and industrial standards [do11, iso11] pose strict requirements for isolating these classes in order to allow *independent certification* of criticality levels

[GSHT13, BFB15, TSP15]. Isolation Scheduling guarantees that tasks of different criticality levels do not interfere on shared platform resources, and therefore, it allows for independent certification as well as a much simplified intra-class interference analysis.

Contribution and Outline. While recent work approached the idea of Isolation Scheduling from different angles (see Section 2.1), we are the first to systematically formalize the IS model (Section 2.5). We specifically analyze the IS model with/without the asymmetric protections among different criticality levels. To this end, we propose fluid-based scheduling techniques and formally study the fundamental loss incurred in the IS model. Our work delivers a deep theoretical understanding about the IS model, and suggests that the IS model is a useful and flexible abstraction for designing scheduling policies for systems that require strong isolation among task classes, such as mixed-criticality systems. Note that the introduction of Section 2.5 and the presentation of Section 2.5.1 are based on our paper publication [HGA+15].

2.5.1 The Isolation Scheduling (IS) Model

The Isolation Scheduling (IS) model dynamically partitions a multi-core processor in time between different task classes so that, at any time, only jobs of the same task class are allowed to execute on the platform. This strategy allows to partition the problem of bounding interference on shared resources to the single task classes; inter-class interference is completely disallowed. We target homogeneous multi-core processors with m identical cores that share on-chip resources. Figure 2.18 shows an example of such an architecture.

The IS model supports real-time tasks τ, where each task periodically or sporadically instantiates single jobs. In addition, we assume that tasks are partitioned into K task classes $S = \{S_k \mid 1 \leq k \leq K\}$, where each task class S_k contains n_k tasks, i.e., $S_k = \{\tau_{i,k} \mid 1 \leq i \leq n_k\}$. For each task $\tau_{i,k}$ in task class S_k, the tuple $(T_{i,k}, D_{i,k}, C_{i,k})$ defines the period of the jobs (or their minimal inter-arrival time), their relative deadline, and their worst-case execution time (WCET). For the analysis in this work, we assume $D \leq T$. However, this is not a necessary constraint for the general Isolation Scheduling model. Additionally, we define the density $\delta_{i,k}$ and utilization $u_{i,k}$ of a task $\tau_{i,k}$ as

$$\delta_{i,k} = C_{i,k}/D_{i,k}, \quad u_{i,k} = C_{i,k}/T_{i,k}. \tag{2.62}$$

Throughout Section 2.5, we use k to index task classes, and i to index tasks within each task class. The set of task classes S is IS-schedulable if all tasks can meet their deadlines while respecting the IS-constraint

of mutual exclusion between task classes. We will refine some of these notations when applying the IS model to mixed-criticality settings.

The IS model treats the multi-core processor as a single resource that needs to be time-partitioned between the different scheduling classes. To give an intuition of how the model works, Figure 2.19 shows an example of an IS schedule.

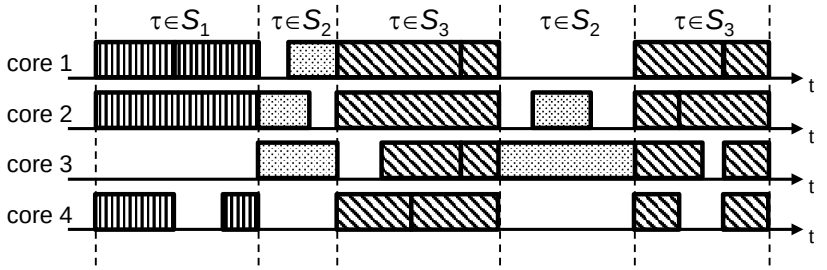


Figure 2.19: Example IS schedule with three task classes S_1, S_2 and S_3. Vertical lines mark the synchronous switching between task classes on all cores.

One way to relate the IS model to single-processor real-time scheduling is by comparing the synchronous task class switches (the vertical lines in Figure 2.19) with classic job preemption. When a single-core processor would switch between jobs, under the Isolation Scheduling model, all cores of the multi-core processor synchronously switch between jobs of two task classes. In other words, classical job preemption is now lifted to synchronous switching between classes. This approach is different from single-core equivalence [SCM+14], which achieves task isolation by individually protecting each shared resource. An alternative way to look at the IS model is from a server perspective. Classical real-time servers [AB98] provide bounded capacities to task sets in a single processor setting. Recent proposals for multi-core servers [SEL08, BBB09] divide a multi-core platform into multiple virtual platforms (servers) that could run in parallel. In Isolation Scheduling, the server acts globally on all cores, i.e., switching the resource allocation happens synchronously on all cores.

The IS model imposes synchronous class switching to bound resource interference, but remains general enough to support different scenarios. For instance, the model supports sporadic and periodic tasks; implicit, constrained, and arbitrary deadlines; preemptive and non-preemptive scheduling; static and dynamic time partitioning; and global and partitioned mapping of tasks to cores. Moreover, the implementation of an IS scheme depends on the specific application. For instance, if the switching between task classes can follow a static schedule, then a time-driven synchronization is appropriate; instead, if switching decisions are determined dynamically at run-time, then global synchronization

mechanisms will be helpful. In Section 2.5.2 to 2.5.3 we analyze the schedulability and propose scheduling algorithms for some of these combinations.

2.5.1.1 Isolation Scheduling for Mixed-Criticality Systems

Mixed-criticality systems are a notable example of systems that require strong isolation between task classes. Mixed-criticality systems are real-time systems where tasks are partitioned into classes, commonly called *criticality levels*; different criticality levels have varying requirements in terms of correctness, assurance, and safety. To allow independent safety certification of criticality levels and avoid costly re-certification, task classes need to be strongly isolated [BFB15, GSHT13, TSP15]. While coarse-grained static partitioning grants such isolation for single-core systems, achieving a similar goal on multi-core systems requires hardware and/or software mechanisms for guaranteeing interference-free or interference-bounded access of tasks to any shared resource, see for example [SCM+14]. Instead, the IS model takes advantage of the existence of well-defined task classes and avoids concurrent execution of jobs with different criticality by construction. In this way, different classes cannot contend on shared resources at all, criticality levels are completely isolated and they can be separately analyzed and certified. We use (particularly, in Section 2.5.3) mixed-criticality systems [Ves07] as a case study to illustrate the usefulness of the IS model.

Recap of the Mixed-Criticality Model. When we discuss the IS model in the context of mixed-criticality systems, we focus on systems with two task classes (i.e., two criticality levels), as commonly assumed for simplicity [BD16]. For convenience, we repeat here the corresponding well-known model (see Section 2.1), with minor notational modifications to fit into the context of Isolation Scheduling.

Task class S_{HI} only consists of tasks of high (HI) criticality; task class S_{LO} only includes LO criticality tasks. The execution time of HI criticality tasks is bounded on both criticality levels: $\forall \tau_i \in S_{\mathrm{HI}}$, the term $C_i(\mathrm{LO})$ denotes the *low execution time bound*, and $C_i(\mathrm{HI})$ denotes the *high execution time bound*. The high execution time bound must be always guaranteed and is assumed to be more pessimistic than the low: $\forall \tau_i \in S_{\mathrm{HI}}$ we require $C_i(\mathrm{HI}) \geq C_i(\mathrm{LO})$. LO criticality tasks only have one execution time bound, i.e., $C_i(\mathrm{HI}) = C_i(\mathrm{LO})$.

Equation (2.63) defines the density $\delta_i(\chi)$ and utilization $u_i(\chi)$ of task τ_i as a function of its execution time bound $\chi \in \{\mathrm{HI}, \mathrm{LO}\}$.

$$\delta_i(\chi) := C_i(\chi)/D_i, \quad u_i(\chi) := C_i(\chi)/T_i. \tag{2.63}$$

Similarly, equation (2.64) defines the density $\Delta_{\chi_1}^{\chi_2}$ and utilization $U_{\chi_1}^{\chi_2}$ of

the task class S_{χ_1} with low (χ_2 = LO) and high (χ_2 = HI) execution time bounds.

$$\Delta_{\chi_1}^{\chi_2} := \sum_{\tau_i \in S_{\chi_1}} \delta_i(\chi_2), \quad U_{\chi_1}^{\chi_2} := \sum_{\tau_i \in S_{\chi_1}} u_i(\chi_2). \tag{2.64}$$

In principle, one could just apply Isolation Scheduling to the above mixed-criticality task model. However, to further improve resource efficiency, asynchronous protections between criticality levels can be enforced, as similarly done in the AIS model: A dual criticality system is in LO *mode* as long as no HI criticality job overruns its low execution time bound C_i(LO); when at least one HI criticality job overruns its low execution time bound, the system switches to HI *mode*. The system must satisfy two schedulability requirements:

- In LO mode, all jobs of LO and HI criticality are schedulable; and
- in HI mode, all HI criticality jobs are schedulable.

After a switch to HI mode, the system can switch back to LO mode under certain circumstances, e.g., when there are no more HI criticality jobs to be scheduled [SGTG12, BCLS14].

2.5.2 The IS-DP-Fair Scheduling Policy

Optimal scheduling for multi-core is provided, for both periodic and sporadic task sets, by the DP-Fair family of algorithms [LFS+10], which originated from P-Fair [BCPV93]. These algorithms allow inter-class contention and inter-class use of shared resources, as discussed in Section 2.1. We tackle this shortcoming by "porting" DP-Fair into the IS model, i.e., we extend DP-Fair with the *IS constraint* of strong isolation between task classes. We call this new algorithm IS-DP-Fair and we present its schedulability analysis, focusing on periodic tasks. We show that, for tasks with implicit deadlines, IS-DP-Fair is optimal in terms of schedulability among all possible schedulers based on the IS model; we quantify the schedulability loss of IS-DP-Fair compared to non-IS schedulers.

2.5.2.1 The IS-DP-Fair Algorithm

Algorithm 2.3 outlines the IS-DP-Fair algorithm. Similarly to DP-Fair, we first partition time into slices: given the sequence of all arrival times and deadlines, a slice σ_j is the time span between two consecutive such instants, with its length denoted as L_j.

As a second step, we subdivide each such slice σ_j into K consecutive subslices; each subslice $\sigma_{j,k}$ of length $L_{j,k}$ (see Algorithm 2.3, step 2) is used to exclusively host task class S_k.

Algorithm 2.3: IS-DP-Fair

1 Let $t_1, t_2, \cdots$ be the sequence of all arrival times and deadlines of the jobs to be scheduled (in increasing order); subdivide time into basic slices σ_j, separated by the t_j time points:

$$\sigma_j = [t_j, t_{j+1}), \text{ with length } L_j.$$

2 Subdivide each σ_j into K subslices: $\sigma_j = \{\sigma_{j,k} | 1 \leq k \leq K\}$. The length of $\sigma_{j,k}$ is

$$L_{j,k} = \max \left\{ L_j \max_{\tau_{i,k} \in S_k} \{\delta_{i,k}\}, \frac{L_j}{m} \sum_{\tau_{i,k} \in S_k} \delta_{i,k} \right\}.$$

3 $\forall i, j, k$ allocate an execution budget $\delta_{i,k} L_j$ for task $\tau_{i,k}$ in subslice $\sigma_{j,k}$.

4 Schedule tasks within each subslice with DP-Fair.

Third, we allocate the execution budget to all tasks. The idea is to enforce proportional progress of all tasks (from different task classes) within each slice: each task $\tau_{i,k}$ is exclusively assigned an execution budget $\delta_{i,k} L_j$ in subslice $\sigma_{j,k}$ within σ_j. Within each subslice $\sigma_{j,k}$, tasks are assigned iteratively in a greedy way: each task is assigned to a processor that has a non-empty (i.e., at least one task was already assigned to it) and non-full (i.e., there is free capacity) subslice $\sigma_{j,k}$. If no such processor exists, the task is assigned to a processor that has empty $\sigma_{j,k}$. If the available capacity in the subslice $\sigma_{j,k}$ for the chosen processor is not enough to serve all the execution budget of task $\tau_{j,k}$, then the budget is split and the excess is allocated to the next available processor. This process continues until all tasks are allocated.

After allocation, we know exactly when and on which processor the budget for each task will be available. At runtime, any task executes whenever its budget becomes available; as long as the task does not arrive, the corresponding budget is idled.

2.5.2.2 Schedulability Analysis of IS-DP-Fair

Theorem 2.9 gives an exact schedulability test for using IS-DP-Fair to schedule a task set S with K task classes S_k, with $1 \leq k \leq K$, on a multi-core architecture with m identical cores.

Theorem 2.9. *Task set S is schedulable with IS-DP-Fair iff*

$$\sum_{S_k \in S} \max \left\{ \max_{\tau_{i,k} \in S_k} \{\delta_{i,k}\}, \frac{1}{m} \sum_{\tau_{i,k} \in S_k} \delta_{i,k} \right\} \leq 1. \tag{2.65}$$

Proof. According to the IS-DP-Fair algorithm, the arrival time t_a and the absolute deadline t_d ($d > a$) of any job of any task $\tau_{i,k}$ from task class S_k coincide with the start or the end of a (possibly different) slice. There might be multiple slices σ_j between t_a and t_d, where $a \leq j < d$. Task $\tau_{i,k}$ is guaranteed to meet its deadline if its execution budget is satisfied across these slices.

Within each slice σ_j, any task $\tau_{i,k}$ receives, by construction, an execution budget of $\delta_{i,k} L_j$ for each subslice $\sigma_{j,k}$ and, since a task cannot run in parallel with itself, we find

$$L_j \max_{\tau_{i,k} \in S_k} \{\delta_{i,k}\} \leq L_{j,k}. \tag{2.66}$$

In addition, since all tasks from S_k must be schedulable by DP-Fair within $\sigma_{j,k}$, i.e., the cumulative allocated budget must be less than or equal to the available processor time from all cores, we conclude

$$L_j \sum_{\tau_{i,k} \in S_k} \delta_{i,k} \leq m L_{j,k}. \tag{2.67}$$

Combining the two bounds from relations (2.66) and (2.67), we get

$$L_{j,k} \geq \max \left\{ L_j \max_{\tau_{i,k} \in S_k} \{\delta_{i,k}\}, \frac{L_j}{m} \sum_{\tau_{i,k} \in S_k} \delta_{i,k} \right\}. \tag{2.68}$$

Setting $L_{j,k}$ to the minimum value that satisfies relation (2.68) guarantees schedulability within each subslice. Finally we consider the additional constraint

$$\begin{aligned} & \sum_{S_k \in S} L_{j,k} \leq L_j \\ \iff & \sum_{S_k \in S} \max \left\{ L_j \max_{\tau_{i,k} \in S_k} \{\delta_{i,k}\}, \frac{L_j}{m} \sum_{\tau_{i,k} \in S_k} \delta_{i,k} \right\} \leq L_j \\ \iff & \sum_{S_k \in S} \max \left\{ \max_{\tau_{i,k} \in S_k} \{\delta_{i,k}\}, \frac{1}{m} \sum_{\tau_{i,k} \in S_k} \delta_{i,k} \right\} \leq 1. \end{aligned} \tag{2.69}$$

Therefore, condition (2.65) is sufficient for schedulability with IS-DP-Fair and, if the test (2.65) fails, then the task set S cannot be scheduled with IS-DP-Fair. Thus, the test as specified in condition (2.65) is exact, i.e., both sufficient and necessary. □

After providing the schedulability test of Theorem 2.9, we show in Theorem 2.10 that the IS-DP-Fair algorithm is optimal for any task set $S = \{S_k\}$ with implicit deadlines under the IS constraint, i.e., under isolation of task classes.

Theorem 2.10. *IS-DP-Fair is optimal in terms of schedulability for task sets with* implicit deadlines *under the IS constraint.*

Proof. We prove Theorem 2.10 by showing that, whenever IS-DP-Fair fails, no other scheduling solution exists. Since the schedulability test of Theorem 2.9 is exact, assuming that IS-DP-Fair fails for task set $S = \{S_k\}$ implies

$$\sum_{S_k \in S} \max \left\{ \max_{\tau_{i,k} \in S_k} \{\delta_{i,k}\}, \frac{1}{m} \sum_{\tau_{i,k} \in S_k} \delta_{i,k} \right\} > 1. \tag{2.70}$$

For the purpose of contradiction, assume that S is still schedulable by some other scheduling algorithm Λ.

Consider the hyperperiod of the tasks of S in the case when all tasks initially arrive at time zero. Let T_{hyper} denote the duration of the hyperperiod and $T_{hyper,k}$ denote the total duration of the subslices allocated to task class S_k within the hyperperiod. A task $\tau_{i,k}$ in S_k cannot run in parallel with itself and it must execute for $\delta_{i,k} T_{hyper}$ within T_{hyper}. Therefore,

$$\begin{aligned} & \forall \tau_{i,k} \in S_k : \quad T_{hyper,k} \geq \delta_{i,k} T_{hyper} \\ \iff & T_{hyper,k} \geq T_{hyper} \max_{\tau_{i,k} \in S_k} \{\delta_{i,k}\}. \end{aligned}$$

Since we assumed that S is schedulable by Λ, then all tasks from S_k meet their deadlines within T_{hyper}, implying

$$\begin{aligned} & m T_{hyper,k} \geq \sum_{\tau_{i,k} \in S_k} \delta_{i,k} T_{hyper} \\ \iff & T_{hyper,k} \geq \frac{T_{hyper}}{m} \sum_{\tau_{i,k} \in S_k} \delta_{i,k}. \end{aligned}$$

Within the hyperperiod, the total fraction of all slices allocated to any task

class cannot be more than T_{hyper}. Therefore:

$$T_{hyper} \geq \sum_{S_k \in S} \max \left\{ \max_{\tau_{i,k} \in S_k} \{\delta_{i,k} T_{hyper}\}, \frac{T_{hyper}}{m} \sum_{\tau_{i,k} \in S_k} \delta_{i,k} \right\}$$

$$\iff \sum_{S_k \in S} \max \left\{ \max_{\tau_{i,k} \in S_k} \{\delta_{i,k}\}, \frac{1}{m} \sum_{\tau_{i,k} \in S_k} \delta_{i,k} \right\} \leq 1. \tag{2.71}$$

Condition (2.70) contradicts condition (2.71), so no algorithm Λ can schedule S. □

While IS-DP-Fair is optimal for implicit deadlines, the same property does not hold for the case of a task set S with constrained deadlines, as Theorem 2.11 states.

Theorem 2.11. *IS-DP-Fair is not optimal for task sets with constrained deadlines under the IS constraint.*

Proof. We prove Theorem 2.11 by showing a counterexample. Consider a task set S with two task classes S_1 and S_2, each containing a single task, respectively $\tau_1 = (2, 1, 1)$[1] and $\tau_2 = (2, 2, 1)$. Trying to schedule S on a dual-core processor fails the schedulability test of Theorem 2.9. However, S is in fact schedulable. Since each class only contains one task, we can reduce the problem to a single core scheduling problem and apply fixed-priority scheduling (τ_1 with higher priority). Then it is straightforward to see, through conventional response time analysis, that S is schedulable. □

2.5.2.3 Schedulability Loss of IS-DP-Fair

Finally, it is important to quantify the loss of schedulability due to enforcing the IS constraint, compared to allowing inter-class interference. Theorem 2.12 provides a tight bound on the speedup required to enforce isolation.

Theorem 2.12. *Any task set S schedulable with DP-Fair (by removing the IS constraint) is schedulable by IS-DP-Fair under the IS constraint on a platform that is* $\min\{K, m\}$ *times faster. This speedup bound is tight.*

Proof. To construct the proof, we first mathematically formulate an optimization problem to find the minimum required speedup of IS-DP-Fair, such that it can schedule any task set schedule by DP-Fair without

[1] The tuple defines the period, relative deadline, and worst-case execution time of the task, see Section 2.1.

the IS constraint. We then reveal properties (Fact 2.1) of our formulated problem, with which a closed-form speedup factor can be derived.

1-Formulating an optimization problem: According to [LFS+10], the following equation gives the schedulability test for DP-Fair:

$$\max\left\{\max_{\substack{\tau_{i,k}\in S_k\\ S_k\in S}}\{\delta_{i,k}\},\ \ \frac{1}{m}\sum_{S_k\in S}\sum_{\tau_{i,k}\in S_k}\delta_{i,k}\right\}\leq 1. \tag{2.72}$$

Now, let us assume that the system is IS-DP-Fair schedulable on a hardware that is λ times faster. Using Theorem 2.9, this implies:

$$\begin{aligned}&\sum_{S_k\in S}\max\left\{\max_{\tau_{i,k}\in S_k}\{\frac{1}{\lambda}\delta_{i,k}\},\frac{1}{m}\sum_{\tau_{i,k}\in S_k}\frac{1}{\lambda}\delta_{i,k}\right\}\leq 1\\ \Longleftrightarrow\ &\lambda\geq\sum_{S_k\in S}\max\left\{\max_{\tau_{i,k}\in S_k}\{\delta_{i,k}\},\frac{1}{m}\sum_{\tau_{i,k}\in S_k}\delta_{i,k}\right\}.\end{aligned} \tag{2.73}$$

Thus the minimum possible speedup bound can be calculated as

$$\begin{aligned}&\lambda_{\min}=\max_{S}\{f(S)\}\\ s.t.\quad&\text{condition (2.72)}\\ \text{where}\quad&f(S)=\sum_{S_k\in S}\max\left\{\max_{\tau_{i,k}\in S_k}\{\delta_{i,k}\},\ \ \frac{1}{m}\sum_{\tau_{i,k}\in S_k}\delta_{i,k}\right\}.\end{aligned} \tag{2.74}$$

2-Properties of the optimization problem (2.74): We will now characterize the the worst case $f(S)$ which maximizes $\lambda_{\min}$ subject to condition (2.72).

Fact 2.1. *$f(S)$ is maximized subject to condition* (2.72) *if:*

$$\forall S_k\in S,\max_{\tau_{i,k}\in S_k}\{\delta_{i,k}\}\geq\frac{1}{m}\sum_{\tau_{i,k}\in S_k}\delta_{i,k}. \tag{2.75}$$

Proof. We prove this by contradiction. Suppose that for some $S_{k'}$, $\max\limits_{\tau_{i,k'}\in S_{k'}}\{\delta_{i,k'}\}<\dfrac{1}{m}\sum\limits_{\tau_{i,k'}\in S_{k'}}\delta_{i,k'}$. We will now compute $\max\limits_{S}\{f(S)\}$ subject to condition (2.72). To maximize $f(S)$ while satisfying condition (2.72), we set:

$$\frac{1}{m}\sum_{\tau_{i,k'}\in S_{k'}}\delta_{i,k'}\leq 1-\frac{1}{m}\sum_{S_k\in S\setminus S_{k'}}\sum_{\tau_{i,k}\in S_k}\delta_{i,k}.$$

As a result, for $S_{k'}$, we have

$$\begin{aligned}\max\left\{\max_{\tau_{i,k'}\in S_{k'}}\{\delta_{i,k'}\}, \frac{1}{m}\sum_{\tau_{i,k'}\in S_{k'}}\delta_{i,k'}\right\} \\ \leq 1-\frac{1}{m}\sum_{S_k\in S\setminus S_{k'}}\sum_{\tau_{i,k}\in S_k}\delta_{i,k}.\end{aligned} \tag{2.76}$$

Now, let us consider two cases:

-$m=1$: In $S_{k'}$, we can have just one single task with density equal to $1-\frac{1}{m}\sum_{S_k\in S\setminus S_{k'}}\sum_{\tau_{i,k}\in S_k}\delta_{i,k}$. This way, $\max_{\tau_{i,k'}\in S'_k}\{\delta_{i,k'}\}=\frac{1}{m}\sum_{\tau_{i,k'}\in S'_k}\delta_{i,k'}$ ($m=1\wedge|S_{k'}|=1$) and (2.72) is still satisfied. Furthermore, the maximum possible value of $\max\left\{\max_{\tau_{i,k'}\in S_{k'}}\{\delta_{i,k'}\}, \frac{1}{m}\sum_{\tau_{i,k'}\in S_{k'}}\delta_{i,k'}\right\}$ is the same compared to calculation (2.76). Therefore, $f(s)$ stays the same.

-$m\geq 2$: Let us denote with 0^+ an infinitely small positive number. Now for $S_{k'}$, we can have just two tasks, one with density $1-\frac{1}{m}\sum_{S_k\in S\setminus S_{k'}}\sum_{\tau_{i,k}\in S_k}\delta_{i,k}+0^+$ and the other with density $1-\frac{1}{m}\sum_{S_k\in S\setminus S_{k'}}\sum_{\tau_{i,k}\in S_k}\delta_{i,k}-0^+$. This way, condition (2.72) is still satisfied while

$$\begin{aligned}\max\left\{\max_{\tau_{i,k'}\in S_{k'}}\{\delta_{i,k'}\}, \frac{1}{m}\sum_{\tau_{i,k'}\in S_{k'}}\delta_{i,k'}\right\} \\ = 1-\frac{1}{m}\sum_{S_k\in S\setminus S_{k'}}\sum_{\tau_{i,k}\in S_k}\delta_{i,k}+0^+,\end{aligned}$$

which is increased compared to calculation (2.76). Thus, $\max_S\{f(S)\}$ gets larger.

Both cases lead to contradictions and condition (2.75) must hold. □

3-Speedup bound: With Fact 2.1, the problem of finding $\lambda_{\min}$ becomes

$$\begin{aligned}&\lambda_{\min}=\max_S\{f(S)\}\\ s.t.\quad &\text{condition (2.72)}\\ &f(S)=\sum_{S_k\in S}\max_{\tau_{i,k}\in S_k}\{\delta_{i,k}\}.\end{aligned} \tag{2.77}$$

To get the maximum possible value of $f(S)$, we have

$$\begin{aligned} f(S) &= \sum_{S_k \in S} \max_{\tau_{i,k} \in S_k} \{\delta_{i,k}\} \\ &\leq \sum_{S_k \in S} 1 \ \text{(from condition (2.72))} \\ &= |S| = K, \end{aligned} \tag{2.78}$$

$$\begin{aligned} f(S) &= \sum_{S_k \in S} \max_{\tau_{i,k} \in S_k} \{\delta_{i,k}\} \\ &\leq \sum_{S_k \in S} \sum_{\tau_{i,k} \in S_k} \delta_{i,k} \leq m \ \text{(from condition (2.72))}. \end{aligned}$$

Thus, $\lambda_{\min} = \min\{K, m\}$.

4-Tightness: Finally, we show that the speedup bound is tight by finding a concrete IS task model that will lead to such a bound: For each $S_k \in S$, let it only contain one single task with density $\frac{\min\{K,m\}}{K} \leq 1$. In this case the total density of the system can be calculated as: $\frac{\min\{K,m\}}{K} \cdot K \leq m$. Thus, condition (2.72) is satisfied and the system is schedulable without the IS constraint by an optimal multiprocessor scheduling algorithm like DP-Fair. Furthermore, we calculate that $f(S) = \frac{\min\{K,m\}}{K} \cdot K = \min\{K, m\}$. As a result, our derived speedup bound is tight. □

For IS task models with implicit deadlines, IS-DP-Fair is optimal according to Theorem 2.10. Therefore, the speedup bound of Theorem 2.12 is optimal in this case. We now extend this result to derive a general bound in case of constrained deadlines using the following theorem.

Theorem 2.13. *Under the IS constraint, no scheduler can achieve a speedup bound better than* $\min\{K, m\}$ *compared to an optimal scheduling algorithm that ignores the IS constraint.*

Proof. In case of constrained deadlines, we can use the same concrete example as shown in the tightness proof for Theorem 2.12. For such an example, if tasks have a zero offset and the same period and deadline, then no IS scheduler is able to schedule it on any platform with speedup less than $\min\{K, m\}$. □

While, in general, there is a cost for enforcing the IS constraint (as Theorem 2.12 and 2.13 show), there is no such cost to pay under suitable assumptions, as Corollary 2.3 shows.

Corollary 2.3. *If a task set S is schedulable by ignoring the IS constraint and if it satisfies the condition*

$$\forall S_k \in S : \quad \frac{\max\limits_{\tau_{i,k} \in S_k} \{\delta_{i,k}\}}{\operatorname{avg}\limits_{\tau_{i,k} \in S_k} \{\delta_{i,k}\}} \leq \frac{|S_k|}{m}, \tag{2.79}$$

where the operator avg *computes the average of its arguments, then S is also schedulable with IS-DP-Fair.*

Proof. By reformatting condition (2.79), we get $\max\limits_{\tau_{i,k} \in S_k} \{\delta_{i,k}\} \leq \frac{1}{m} \sum\limits_{\tau_{i,k} \in S_k} \delta_{i,k}$. If the system is schedulable without the IS constraint and condition (2.79) holds, then we have:

$$f(S) = \frac{1}{m} \sum_{S_k \in S} \sum_{\tau_{i,k} \in S_k} \delta_{i,k} \leq 1 \;\; (f(S) \;\; \textit{given by formulation} \;\; (2.74)). \tag{2.80}$$

According to Theorem 2.9, the system is IS-DP-Fair schedulable. □

Essentially Corollary 2.3 states that the schedulability loss due to isolation decreases, as the variation in density across tasks within task classes decreases and, as the number of tasks within task classes increases.

2.5.3 IS-DP-Fair for Mixed-Criticality Systems

An immediate application of IS-DP-Fair is scheduling mixed-criticality systems on multi-core. Thanks to the IS model, IS-DP-Fair ensures isolation between criticality levels. While all the results of Section 2.5.2 apply, mixed-criticality systems have some additional peculiarities (see Section 2.1), e.g., they can switch between criticality modes. For this reason, we extend IS-DP-Fair for mixed-criticality systems; we call this new algorithm MC-IS-Fluid. An extension of the DP-Fair algorithm for mixed-criticality systems exists for multiprocessors without considering isolation. We outlined this algorithm, known as MC-Fluid [LPG+14], in Section 2.1.1. The key idea, borrowed from [BBD+12], is to shorten the deadlines of HI criticality tasks in LO mode, in order to shift demand from HI to LO mode and improve schedulability. We adopt the same technique in MC-IS-Fluid.

2.5.3.1 The MC-IS-Fluid Algorithm

Algorithm 2.4 outlines the MC-IS-Fluid algorithm for a dual-criticality task set $S = \{S_{\text{HI}}, S_{\text{LO}}\}$. We compute shortened deadlines for HI criticality

Algorithm 2.4: MC-IS-Fluid

1 For all HI criticality tasks $\tau_i \in S_{\text{HI}}$, compute the shortened deadline $D'_i = xD_i$, to be used in LO mode; the shortening factor $x : 0 < x \leq 1$ is:

$$x = \frac{\max\left\{\max_{\tau_i \in S_{\text{HI}}}\{\delta_i(\text{LO})\}, \quad \frac{1}{m}\Delta_{\text{HI}}^{\text{LO}}\right\}}{1 - \max\left\{\max_{\tau_i \in S_{\text{LO}}}\{\delta_i(\text{LO})\}, \quad \frac{1}{m}\Delta_{\text{LO}}^{\text{LO}}\right\}}. \tag{2.81}$$

2 In LO mode, set the density of all HI criticality tasks $\tau_i \in S_{\text{HI}}$ to $\delta_i(\text{LO})/x$; for LO criticality tasks $\tau_i \in S_{\text{LO}}$ use the density $\delta_i(\text{LO})$. In LO mode, schedule all tasks by IS-DP-Fair.

3 If any HI criticality task overruns its LO level WCET, first conclude the current slice; then, switch to HI mode by terminating all LO criticality tasks and restoring the original deadlines of HI criticality tasks. Schedule the remaining HI criticality tasks with DP-Fair (densities for HI criticality tasks set by Lemma 2.13).

tasks, similarly to MC-Fluid. In LO mode, we schedule the system with IS-DP-Fair, using the original deadlines for LO criticality tasks and the shortened deadlines for HI criticality tasks. After a switch to HI mode, all LO criticality tasks are dropped and we schedule the remaining HI criticality tasks with a DP-Fair compatible scheduling technique, using the original deadlines. While using a similar approach to MC-Fluid, MC-IS-Fluid has two main differences:

- we shorten the deadlines of HI criticality tasks uniformly, so we are able to provide a closed-form schedulability test; and
- we show (see Lemma 2.14) that greedily shortening the deadlines of HI criticality tasks is optimal for schedulability in HI mode.

2.5.3.2 Schedulability Analysis for MC-IS-Fluid

In LO mode, MC-IS-Fluid uses shortened deadlines $D'_i = xD_i$, with $0 < x \leq 1$ (see Algorithm 2.4), for all HI criticality tasks $\tau_i \in S_{\text{HI}}$. Therefore, the density of each task $\tau_i \in S_{\text{HI}}$ in LO mode increases to $\delta_i(\text{LO})/x$. MC-IS-Fluid does not shorten the deadlines of LO criticality tasks; so, $\forall \tau_i \in S_{\text{LO}}$, the density is $\delta_i(\text{LO})$. We can use the schedulability test of Theorem 2.9, by simply using the shortened deadlines for HI criticality tasks, to test schedulability in LO mode.

Finding a closed-form schedulability test for MC-IS-Fluid in HI mode

is less trivial because, in general, we do not know when the system will switch to HI mode. If, at mode switch, a partially executed HI criticality job is *carried over* to HI mode, we need to know the remaining execution requirement of this job and the time until its actual deadline in order to bound the maximum task density in HI mode. Let us denote such maximum density of a HI criticality task τ_i in HI mode as $\delta_i^{\max}(\text{HI})$. Formally, we establish the following result.

Lemma 2.13. *For any* HI *criticality task* τ_i,

$$\delta_i^{max}(\text{HI}) = \max\left\{\frac{\delta_i(\text{HI}) - \delta_i(\text{LO})}{1-x}, \delta_i(\text{HI})\right\}. \tag{2.82}$$

Proof. We define a carry-over job as one job that starts in LO mode but finishes in HI mode.

Consider $\tau_i \in S_{\text{HI}}$. If there is no carry-over job from this task in HI mode, then we only need to consider jobs of τ_i that arrive after the mode switch. In this case the maximum density of τ_i in HI mode equals $\delta_i(\text{HI})$.

Otherwise, we need to consider the carry-over job. Assume that the mode switch happens $t^\star$ after the arrival of a job of τ_i, where $0 \leq t^\star \leq xD_i$. Since, according to Algorithm 2.4, the mode switch coincides with the end of one slice in LO mode, then the density of this carry-over job in HI mode is:

$$\delta_i^*(\text{HI}) = \frac{C_i(\text{HI}) - \frac{\delta_i(\text{LO})}{x}t^\star}{D_i - t^\star}. \tag{2.83}$$

Since t^* can only vary in the interval $[0, xD_i]$, we need to find the maximum of $\delta_i^*(\text{HI})$ within this interval. To do so, in equation (2.84) we compute $d\,\delta_i^*(\text{HI})/dt^\star$, i.e., the first order derivative of $\delta_i^*(\text{HI})$ with respect to t^*:

$$\begin{aligned}
\frac{d\,\delta_i^*(\text{HI})}{dt^\star} &= \frac{d\left(\frac{C_i(\text{HI}) - \frac{\delta_i(\text{LO})}{x}t^\star}{D_i - t^\star}\right)}{dt^\star} \\
&= \frac{1}{(D_i - t^\star)^2}\left(-\frac{\delta_i(\text{LO})}{x}(D_i - t^\star) + C_i(\text{HI}) - \frac{\delta_i(\text{LO})}{x}t^\star\right) \\
&= \frac{1}{(D_i - t^\star)^2}\left(-\frac{\delta_i(\text{LO})}{x}D_i + C_i(\text{HI})\right).
\end{aligned} \tag{2.84}$$

Looking at equation (2.84), the sign of $d\,\delta_i^*(\text{HI})/dt^\star$ does not change within the interval $[0, xD_i]$. Therefore, the maximum of $\delta_i^*(\text{HI})$ will be at one of the extremes of the interval, according to whether $\delta_i^*(\text{HI})$ is increasing or decreasing (respectively, if $d\,\delta_i^*(\text{HI})/dt^\star$ is positive or negative). Since the sign of equation (2.84) is determined by the second factor, we can look at two cases:

- When $\delta_i(\text{LO})/x \leq \delta_i(\text{HI})$, equation (2.84) is non-negative and the maximum density of the carry-over is when $t^\star = xD_i$:

$$\begin{aligned}&\max_{t^\star = xD_i}\left\{\delta_i^*(\text{HI})\right\}\\ =&\frac{C_i(\text{HI}) - \frac{\delta_i(\text{LO})}{x}xD_i}{D_i - xD_i} = \frac{\delta_i(\text{HI}) - \delta_i(\text{LO})}{1-x}\\ \geq& \frac{\delta_i(\text{HI}) - x\delta_i(\text{HI})}{1-x} = \delta_i(\text{HI}).\end{aligned} \tag{2.85}$$

- When $\delta_i(\text{LO})/x > \delta_i(\text{HI})$, equation (2.84) is negative and the maximum density of the carry-over is when $t^\star = 0$:

$$\begin{aligned}&\max_{t^\star = 0}\left\{\delta_i^*(\text{HI})\right\}\\ =&\delta_i(\text{HI}) = \frac{\delta_i(\text{HI}) - x\delta_i(\text{HI})}{1-x} > \frac{\delta_i(\text{HI}) - \delta_i(\text{LO})}{1-x}.\end{aligned} \tag{2.86}$$

Putting the two cases together, we get Lemma 2.13. □

Using Lemma 2.13, we can formally determine a schedulability test for HI mode (Theorem 2.14).

Theorem 2.14. *A dual-criticality task set $S = \{S_{\text{HI}}, S_{\text{LO}}\}$ is schedulable in* HI *mode under MC-IS-Fluid if*

$$\max\left\{\max_{\tau_i \in S_{\text{HI}}}\{\delta_i^{max}(\text{HI})\},\ \frac{1}{m}\sum_{\tau_i \in S_{\text{HI}}}\delta_i^{max}(\text{HI})\right\} \leq 1. \tag{2.87}$$

Proof. Theorem 2.14 directly follows from Lemma 2.13 and the DP-Fair schedulability test [LFS+10]. □

2.5.3.3 Optimal Greedy Choice of x

So far, our analysis did not assume any particular choice of x. With Lemma 2.14, we show now that setting x according to equation (2.81) in MC-IS-Fluid is optimal in terms of schedulability.

Lemma 2.14. *Whenever a dual-criticality task $S = \{S_{\text{HI}}, S_{\text{LO}}\}$ is schedulable with MC-IS-Fluid for some choice of x, it is also schedulable when x is set according to equation* (2.81).

Proof. Based on Theorem 2.9, in order to guarantee schedulability under IS-DP-Fair in LO mode, we have:

$$\begin{aligned} &\frac{1}{x}\max\left\{\max_{\tau_i\in S_{\mathrm{HI}}}\{\delta_i(\mathrm{LO})\}, \frac{\Delta_{\mathrm{HI}}^{\mathrm{LO}}}{m}\right\} \\ &+\max\left\{\max_{\tau_i\in S_{\mathrm{LO}}}\{\delta_i(\mathrm{LO})\}, \frac{\Delta_{\mathrm{LO}}^{\mathrm{LO}}}{m}\right\} \leq 1. \end{aligned} \tag{2.88}$$

Now, suppose that there exists some x' which leads to a schedulable system in both LO and HI modes. Then, in order to guarantee LO mode schedulability, we must have that $x' \geq x$. According to Lemma 2.13 and Theorem 2.14, if we choose x instead of x', then the maximum task density in HI mode will not increase and the system remains schedulable in HI mode. Therefore, setting x according to equation (2.81) is optimal. □

Finally, with Theorem 2.15, we summarize our analysis into a complete schedulability test.

Theorem 2.15. *A dual-criticality task $S = \{S_{\mathrm{HI}}, S_{\mathrm{LO}}\}$ is schedulable with MC-IS-Fluid under the IS constraint if $0 < x \leq 1$, with x set by equation* (2.81)*, and condition* (2.87) *is satisfied.*

Proof. Theorem 2.15 directly follows from Theorem 2.9, Theorem 2.14 and Lemma 2.14. □

2.5.4 Experimental Evaluation

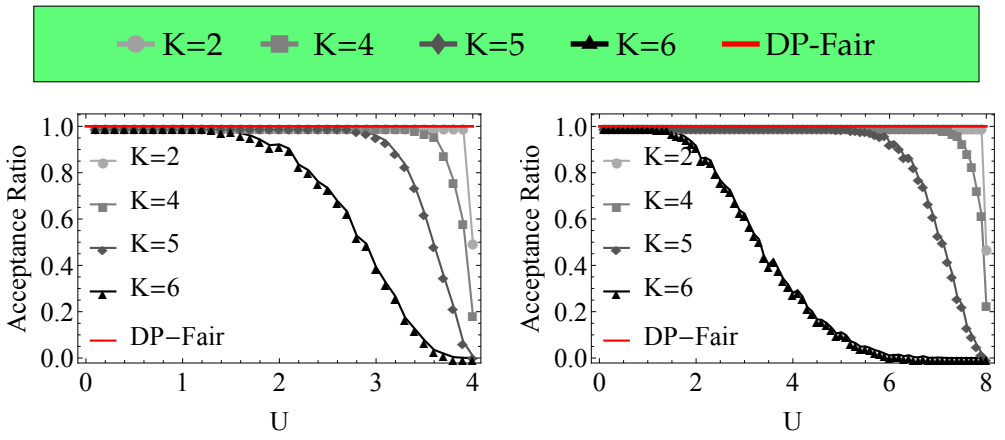
In the following, we evaluate the performance of IS-DP-Fair and MC-IS-Fluid (Section 2.5.2 and 2.5.3) in terms of schedulability. Additionally, we provide comparisons with state-of-the-art scheduling techniques.

2.5.4.1 Random Task Set Generation

We synthetically generate implicit-deadline periodic task sets at different system utilization points. For creating basic (non mixed-criticality) IS task classes, we generate tasks in the following manner:

- Periods are randomly chosen from $\{T \in \mathbb{Z} \mid 2 \leq T \leq 2000\}$.
- Task utilizations are uniformly chosen from $[0.02, 0.2]$.
- Tasks are equally likely to belong to task class $\{S_1, \cdots, S_K\}$.
- Total system utilization is defined as $U := \sum \frac{C}{T}$.

For creating dual-criticality, implicit-deadline task sets, we adopt the task generator as shown in Section 2.3.3.1. We recap here important parameters and present their configurations:



(a) IS-DP-Fair: Impact of number of classes on schedulability, 4 cores

(b) IS-DP-Fair: Impact of number of classes on schedulability, 8 cores

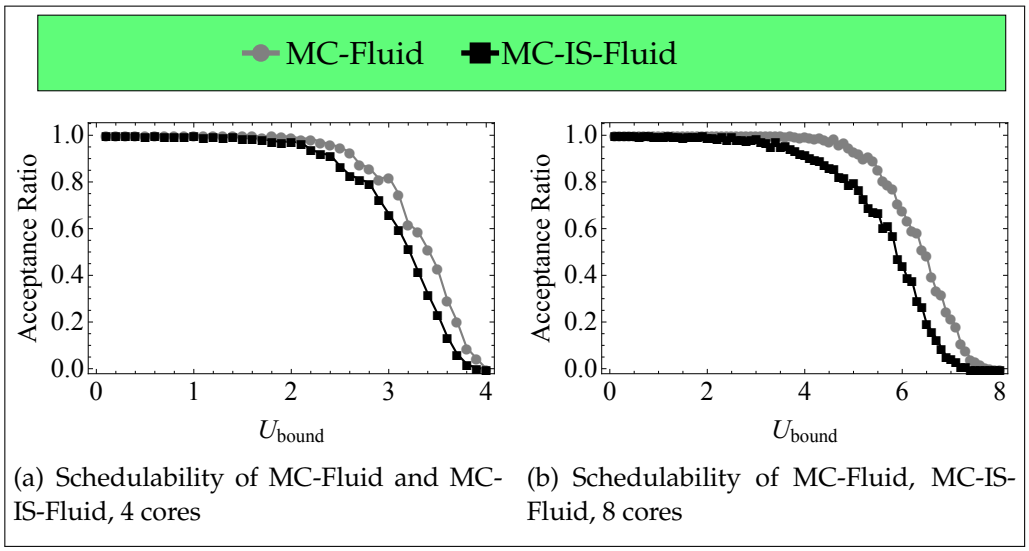
Figure 2.20: Isolation Scheduling: Fraction of schedulable task sets vs. system utilization. Red lines in plots correspond to DP-Fair.

- Probability of any task being HI criticality $P_{\text{HI}} = 0.2$.
- $r = C_i(\text{HI})/C_i(\text{LO})$; for each HI criticality task r is chosen uniformly from $[1, 5]$.
- The utilization U_{bound} of any dual-criticality task set is defined as $\max\{U_{\text{HI}}^{\text{LO}} + U_{\text{LO}}^{\text{LO}}, U_{\text{HI}}^{\text{HI}}\}$.

Periods and LO level utilizations for dual criticality task sets are generated similar to the non mixed-critical task sets. For both basic IS task sets and dual criticality task sets, we perform experiments with system utilizations varying in the interval $[0.1, 4]$ (quad-core experiments) or $[0.1, 8]$ (octa-core experiments). Utilization is incremented in steps of 0.1.

2.5.4.2 Schedulability

Isolation Scheduling. First, we evaluate the schedulability loss caused by enforcing the IS constraint (i.e., mutual exclusion among task classes). For this purpose, we generate 1000 task sets for each system utilization and compute the fraction of task sets that are deemed schedulable under the IS-DP-Fair approach on $m = 4$ and $m = 8$ cores. The results are depicted in Figure 2.20(a) (4 cores) and Figure 2.20(b) (8 cores). For both configurations, the acceptance ratio by IS-DP-Fair decreases as the number of classes increases. This is intuitive: with only one task class, IS-DP-Fair is equivalent to DP-Fair and hence optimal. By adding more classes, we limit task parallelism within each class, thus



(a) Schedulability of MC-Fluid and MC-IS-Fluid, 4 cores

(b) Schedulability of MC-Fluid, MC-IS-Fluid, 8 cores

Figure 2.21: Isolation Scheduling integrated with AIS.

impairing schedulability. Our results here also match with the theoretical analysis. According to Theorem 2.13, with increasing number of classes or processors, the speedup bound of IS-DP-Fair to catch DP-Fair increases, i.e., schedulability decreases.

Isolation Scheduling integrated with AIS. We now present results for dual-criticality systems and compare our approaches to a state-of-the-art scheduling technique MC-Fluid [LPG+14]. We present our results in Figure 2.21(a) (4 cores) and Figure 2.21(b) (8 cores). As shown in the figures, the feasibility of MC-IS-Fluid is very close to MC-Fluid for all utilizations. Therefore, we conclude that for dual-criticality task systems, the cost of enforcing Isolation Scheduling by MC-IS-Fluid is relatively low. However, as explained for Figure 2.20(a), this cost is expected to increase as the number of criticality levels increases.

2.5.5 Summary of Results

Section 2.5 formalized recent advances in isolation for mixed-criticality systems into the Isolation Scheduling model, where the hardware platform can only be accessed by one task class/criticality at one time; thus, inter task class/criticality interferences are excluded by construction. Optimal scheduling technique was proposed and the limit of the new scheduling model was studied. Asymmetric protections among different criticality levels were further integrated into the proposed techniques to achieve runtime adaptiveness and resource efficiency.

2.6 Summary

Specifying the service guarantees as well as their conditions for different criticality levels is a central problem for mixed-criticality systems. Current results mainly follow two directions – they either mix tasks of different criticality levels on the same computing platform and advocate asymmetric protections among those criticality levels, or build upon conventional isolation based models with further extensions. In the former case, the system is advocated to react adaptively to runtime threats (e.g., task overrun) by sacrificing less critical tasks and protecting only critical ones. This, while being effective in improving system resource efficiency, could greatly impair system services or even safety (see Chapter 3).

The first contribution of this chapter is the introduction of three different mixed-criticality models as well as corresponding scheduling techniques, enabling flexible and practical modeling of mixed-criticality systems. The first service adaptation model introduces explicit guarantees for less critical tasks under urgent scenarios (i.e., degraded service and service resetting time). The second model introduces a graphical representation of allowed interferences among tasks of different criticality levels. This model (called ICG) is shown to be more expressive than conventional mixed-criticality models; furthermore, it allows systematic optimization of system interferences thanks to its graphical representation. The last over-clocking model allows the system to explore hardware features to improve the service guarantees for mixed-criticality systems. For all introduced models, accompanying scheduling techniques are developed as case studies of those models.

The second contribution of this chapter is the formalization and analysis of recent works on global temporal isolation among different criticality levels. The model, called Isolation Scheduling, constructively removes interferences on all shared resources among different criticality levels (or task classes), by allowing only one level/class to access the platform at one time. We propose optimal scheduling techniques and formally study the limits of the Isolation Scheduling model. Extensions to integrate with asymmetric isolation among different levels are also studied to further improve the system resource efficiency.

3

Mixed-Criticality Fault-Tolerance – Methods, Analysis and Findings

Threats that could potentially hamper correct system functioning are necessary concerns in the design of (mixed) safety-critical systems; they often arise from diverse sources (e.g., software faults, hardware faults, task overrun, system overheating, design errors, etc.), and must be controlled or mitigated in order to guarantee the system safety. In the context of mixed-criticality systems, the possible consequences of threats tend to get worse, as malfunctioning of less critical tasks could affect other critical tasks through shared resources; this, however, would have been avoided if functionalities of different criticality levels are strictly isolated. As an example, we have discussed in Chapter 2 the scenario that an overrun of a less critical task can delay the execution of other critical tasks, ultimately causing them to miss their deadlines. Furthermore, due to the mixed-criticality nature, different safety assurances are expected on different criticality levels.

To facilitate the design of mixed-criticality systems, the industry often follows well-established safety standards, e.g., DO-178B [DO-92] for avionics and ISO 26262 [iso11] for automotive, with the goal to guarantee *both* functional and non-functional (real-time) safety requirements on all criticality levels. For functional safety, various stresses (e.g., hardware/software errors) [HYT14a] need to be mitigated through various hardening techniques including task re-execution and replication [BDMS+11, HRH+12]. For real-time guarantees, while conventional techniques favor strict temporal and spatial isolation among varying criticality levels [MEA+10, GSHT13], recent advances in mixed-

criticality advocate asymmetric isolation (AIS) among them [BD16] – whenever timing threats (task overruns) are detected, less critical tasks are degraded and their occupied resources are freed to guarantee more critical tasks.

Related Work & Motivation: To a large extent, guaranteeing functional safety and satisfying task deadlines have been studied in isolation for mixed-criticality systems in the literature [BD16]. Making the above two guarantees on a commercial-off-the-shelf (COTS) platform is still not much explored. To date, the mixed-criticality community has mainly focused on the scheduling aspect of such a challenging problem [BD16], e.g., scheduling techniques and schedulability analysis [Ves07, BV08, BBD+12, BCLS14, LB10, EY12, SGTG12, PK11], communication methods and analysis [BD+13, BHI14], and interference analysis with respect to shared resources [GSHT13, YYP+12].

Other works, though trying to address both safety and schedulability together [BDMS+11, HRH+12, KYK+14], might incur some drawbacks: They do not model safety explicitly according to common safety standards, see e.g., DO-178B [DO-92] and ISO 26262 [iso11]. Furthermore, mixed-criticality scheduling techniques commonly advocate runtime adaptations in urgent scenarios, e.g., by dropping less critical tasks or degrading their services [BD16]. To our best knowledge, the impact of such reconfigurations on system safety has not been considered for single-core/multi-core platforms.

Challenges: We tackle in this chapter mixed-criticality fault-tolerant scheduling on homogeneous multi-core platforms (with single-core as a special case), where both system safety and schedulability requirements must be satisfied. Focusing on hardware/software transient errors and dual-criticality systems, task re-execution (on each core) and replication (on multiple cores) are considered as the adopted fault-tolerance mechanisms. It turns out that such a problem is highly non-trivial, as the following challenges is involved:

1 A system level model needs to be defined in the first place to specify the desired system behavior. One essential question is involved: Under common fault tolerance mechanisms, when should runtime adaptation be triggered to enforce asymmetric protections among different criticality levels?

2 A formal analysis framework is required to analyze the system safety given a high level behavior model; this is intrinsically a difficult problem, as in general we do not know when threats and adaptations occur at runtime, which will affect the system safety.

3 A scheduling framework needs to be designed to jointly consider fault-tolerance, run-time adaptation and the real time requirements of tasks, while respecting the different safety requirements on different criticality levels.

Contribution: For the mixed-criticality fault-tolerance problem on multi-core platforms, we provide a complete framework to perform system modeling, safety analysis and scheduling, such that both system safety and schedulability can be satisfied. In detail, our contributions can be summarized as follows:

- We explicitly model system safety requirements on any criticality criticality level by the corresponding pfh (probability-of-failure-per-hour), as commonly used in safety standards [DO-92, Bro00]. This approach could finally enable the designed techniques to be compliant with industrial standards.
- We propose system reconfigurations to enhance resource efficiency – when critical tasks do not succeed after a certain number of trials, less critical tasks are refrained from execution to guarantee the critical tasks. We formulate the fault-tolerance problem using redundancy and adaptation profiles, where task re-execution, task replication and system reconfigurations are considered jointly to achieve a feasible design. We then propose a problem transformation, leveraging classical mixed-criticality scheduling techniques to guarantee task deadlines.
- We develop diverse analysis techniques to bound system safety when reconfigurations are triggered on each core locally or on all cores globally; this enables the trade-off between analysis complexity and system feasibility. To our best knowledge, those are the first results that bound system safety in mixed-criticality systems with runtime adaptation. Our analysis also sheds light on the intrinsic trade-off between system safety and schedulability.
- We validate our proposed techniques with a realistic flight management system [HYT14a] and synthetic task sets, where the impact of online reconfigurations on the overall system feasibility is evaluated. Our results reveal quantitatively that service degradation for mixed-criticality systems can greatly improve system feasibility, while task killing only helps if performed on non-safety related tasks.

Organization: The remainder of this chapter is organized as follows. In Section 3.1, we introduce the background and define our studied problem. Section 3.2 and Section 3.3 describe two diverse analysis techniques to bound system safety. Section 3.4 presents our evaluation results while Section 3.5 concludes this chapter.

Table 3.1: Important notations in Chapter 3

$C_{ik}(\chi)$	WCET of task τ_i on criticality level χ on core π_k
$\tau(\chi)$	tasks with criticality χ
τ^k	tasks executed on core π_k
$\tau^k(\chi)$	all tasks with criticality level χ on core π_k
$N : \tau \times P \rightarrow \mathbb{N}$	Redundancy Profile (Definition 3.1)
n_{ik}	abbreviation of $N(\tau_i, \pi_k)$
n_i	$n_i = \sum_{k=1}^{\lvert\pi\rvert} n_{ik}$
$\text{pfh}(\chi)$	probability of failure per hour of tasks with criticality level χ
f_i	probability that any instance of task τ_i fails with C_i units of execution
d_f	service degradation factor
$N' : \tau(\text{HI}) \times P \rightarrow \mathbb{N}$	Adaptation Profile (Definition 3.2)
n'_{ik}	abbreviation of $N'(\tau_i, \pi_k)$
$\Gamma(\tau, P, N(\cdot), N'(\cdot))$	constructed mixed-criticality task set of τ based on its $N(\cdot)$ and $N'(\cdot)$.

3.1 Background & Problem Statement

We present in this section some background on mixed-criticality systems and scheduling. We subsequently introduce our fault models, common safety requirements as found in [DO-92, Bro00] and corresponding fault-tolerance mechanisms. We finally present a concrete problem definition. All important notations used in this chapter can be found in Table 3.1.

3.1.1 Mixed-Criticality Task Model

We first briefly introduce our assumed task model in this chapter. We consider dual-criticality systems; one such system consists of $|\tau|$ independent sporadic tasks $\{\tau_i | \tau_i \in \tau\}$ running on $|\pi|$ identical cores $\pi = \{\pi_1, \pi_2, ..., \pi_{|\pi|}\}$. Each task is characterized by a 4-tuple $\tau_i = (T_i, D_i, C_i, \chi_i)$, where T_i is the minimal inter-arrival time ($T_i \ll 1$ hour) and D_i is the relative deadline ($D_i \leq T_i$). Each instance of τ_i takes a constant execution time C_i to finish at runtime. χ_i is the criticality level (being either high (HI) or low (LO)) of τ_i. For notational convenience, we use $\mathcal{X}$ to denote the set of existing criticality levels and $\tau(\chi)$ to represent all tasks with criticality level $\chi \in \mathcal{X}$. Our assumed model is simpler than the one presented in Section 2.1, since we do not consider task overrun in this work. We shall explain in the rest of this chapter how to transform the fault-tolerant

Table 3.2: DO-178B safety requirements

χ	A	B	C	D	E
pfh	$< 10^{-9}$	$< 10^{-7}$	$< 10^{-5}$	$\geq 10^{-5}$	–

mixed-criticality scheduling problem (Definition 3.6) into a conventional mixed-criticality scheduling problem as shown in Section 2.1.

We assume that all tasks run for a system operation duration of O_s hours, after which the system state is reset. For example, the typical system operation duration would be approximately 2 hours for a domestic flight and 10 hours for a long international flight.

3.1.2 Fault Model and Safety Requirements

Due to intrinsic transient hardware/software errors [HYT14a], for any instance of task τ_i, we assume there is a probability that it does not finish successfully, denoted as f_i. We assume that failures of all task instances issued by all tasks are independent of each other.

Because of the potential faults, safety measures need to be applied to indicate how safe a system is. We use the probability-of-failure-per-hour (pfh), which is widely adopted in safety standards [DO-92, Bro00]; pfh can be efficiently calculated as the system failure rate in an hour [HGST13, Bro00]. Naturally, the exact calculation of pfh is related to the system level on which this safety measure is required. The pfh of one single task involves all instances of this task in one hour, while the pfh of a task set further requires considering all tasks. According to the general safety standard IEC 61508 [Bro00], pfh is often specified on each criticality level. In this case, our assumption is that it represents the probability that at least one task instance of this criticality fails in one hour. For a system operation of O_s hours, we represent pfh using its average value during the system operation.

In this chapter, we follow the DO-178B [DO-92] safety standard, which defines 5 criticality levels (A is the highest and E is the lowest). The corresponding safety requirements are shown in Table 3.2. As we can see, pfh decreases with increasing χ, i.e., safety requirements are more stringent on higher criticality levels. We say that a mixed-criticality system is safe if the pfh requirements on all criticality levels are satisfied, and it is the interest of this chapter to upper-bound pfh of each criticality level. The results of our work can be applied to a system that contains any two criticality levels in DO-178B.

3.1.3 Fault Tolerance

To mitigate errors and to enhance system safety, two established approaches exist. First, one could detect such errors at runtime by performing sanity-checks [BDMS+11]. We suppose that at the end of a job's WCET, it is known whether the job has failed or not; a faulty job is re-executed in the hope that it will succeed. Second, one could replicate the job a priori on multiple processors, enhancing the chance that one of them will succeed. We assume both in this work: Each instance of any task is replicated on multiple cores offline due to safety requirements, where migrating task instances from one core to another at runtime is forbidden due to high runtime overheads [MOP10]; in case faults are detected on one core, the faulty task instance is re-executed locally up to a given number of times to achieve the required safety. We assume that all replicas of the same task instance fail independently. Note that not all tasks need to have replicas. If a replicate of one task's instance succeeds on some core, replicates on the other cores need not to proceed. However, for our safety analysis, we need to consider the worst-case (all replicas of each task instance fail) to bound the failure probability.

We proceed to introduce some notations. We use τ^{k} to represent all tasks mapped to core π_{k}, and assume any instance of task τ_{i} can execute at most n_{ik} times on core π_{k} and at most n_{i} times on all cores, such that $\sum_{k=1}^{K} n_{\mathrm{ik}} = n_{\mathrm{i}}$, where we call n_{ik} the re-execution profile of task τ_{i} on core π_{k}. Formally, we can define a function N to jointly represent task replication and re-execution (referred to as the system redundancy profile), for all LO and HI criticality tasks.

Definition 3.1 (Redundancy Profile)**.** *We define the redundancy profile* $N : \tau \times P \rightarrow \mathbb{N}$*, where* $N(\tau_i, \pi_k)$*, abbreviated as* n_{ik}*, is the maximum number of executions of any instance of task* τ_i *on core* π_k*.*

With the redundancy profile, we have a general problem formulation: if a task τ_{i} is not replicated on core π_{k}, then $n_{\mathrm{ik}} = 0$; if a task τ_{i} has no replicas, then $n_{\mathrm{i}} = 1$. Furthermore, given the redundancy profile, one needs to solve consequently two sub-problems: 1- All tasks with their replicas need to be scheduled on a multi-core platform while ensuring task deadlines. 2- The safety requirements of all criticality levels need to be satisfied given the scheduling. Afterwards, it is subject to the system safety and schedulability analysis to find the feasible redundancy profile, i.e. the one which meets both safety and schedulability requirements. We will elaborate this in the rest of this chapter.

For notational convenience, we use $N_{\mathrm{k}}(\chi) = \{n_{\mathrm{ik}} \mid \tau_{\mathrm{i}} \in \tau^{\mathrm{k}} \wedge \chi_{\mathrm{i}} = \chi\}$ to denote the redundancy profiles of all χ criticality tasks on core π_{k}.

3.1.4 System Reconfiguration

Real-time embedded systems are typically resource constrained, while task replication and re-execution are costly in terms of required resources. To reallocate resources occupied by less critical tasks to more critical ones when there is an urgency (i.e., when any critical task instance does not succeed after a certain number of trials), a straightforward approach is to drop less critical tasks or degrade their services. We note that the embedded system industry is already investigating/adopting such runtime adaptations in safety-critical or mixed-criticality systems, see e.g., [ZPKW13, saf16].

However, this would naturally affect the safety of the less critical tasks and might only work if those tasks are not safety relevant, e.g., LO = E as shown in Table 3.2. Otherwise, the impact of system reconfigurations on safety needs to be addressed.

We show in the following an example system on a single-core to better explain the need of reconfiguration and its implications.

Table 3.3: Example 3.1 task set

τ	τ_1	τ_2	τ_3	τ_4	τ_5
χ	HI	HI	LO	LO	LO
T/D	60	25	40	90	70
C	5	4	7	6	8

Example 3.1. *Consider a set of 5 sporadic tasks as shown in Table 3.3 with task parameters in units of* ms*, to be scheduled on a single-core. The tasks have criticality levels* HI *and* LO*, with* HI $\in$ {A,B,C} *and* LO $\in$ {D,E}*. The failure probability of each job for every task is assumed to be* 10^{-5}*.*

Since LO *criticality tasks are either level D or level E tasks, the pfh of* LO *criticality is of no interest and we can set* $n_3 = n_4 = n_5 = 1$*. For* HI *criticality tasks, we can search using our safety analysis in Section 3.2 their minimal re-execution profiles:* $n_1 = n_2 = 3$*. In this case the calculated pfh of* HI *criticality is* 2.04×10^{-10}*, which satisfies the safety requirement on the* HI *criticality level. This will lead to an unschedulable system as the the total system utilization is greater than 1:* $3 \times \sum_{\tau_i \in \tau_{\text{HI}}} \frac{C_i}{T_i} + \sum_{\tau_i \in \tau_{\text{LO}}} \frac{C_i}{T_i} = 1.08595 > 1$.

However, since there is no requirement on the upper bound of pfh for level D/E tasks, they can be dropped without jeopardizing the system safety. Hence, to guarantee the schedulability of HI *criticality tasks, one may drop* LO *criticality tasks, e.g., when any* HI *criticality task instance executes a third time. This will indeed make the task set schedulable with the fault-tolerant scheduling technique in Section 3.1.5.*

Notice that, if LO *criticality level is safety relevant, then analysis techniques should be developed to bound the impact of system reconfiguration on the safety of* LO *criticality level.*

To formally model reconfiguration on each core, we use $\tau^{k}(\chi)$ to denote all χ criticality tasks executed on core π_{k}. For $\tau_{i} \in \tau^{k}(\text{HI})$, we assume that dropping LO criticality tasks or degrading their services is controlled by the parameter n'_{ik} ($n'_{ik} \in \mathbb{N} \wedge n'_{ik} \leqslant n_{ik}$): If no instance of any τ_i executes for the $(n'_{ik}+1)$th time on π_{k}, we say π_{k} is in a normal mode and all local tasks can be guaranteed; otherwise, we say it is in an urgent mode and system reconfigurations must be performed. We call n'_{ik} the adaptation (killing or service degradation) profile of any HI criticality task τ_{i} on core π_{k}. We can now define a function N' to specify the system adaptation profile for all HI criticality tasks.

Definition 3.2 (Adaptation Profile)**.** *We define the adaptation profile* $N' : \tau(\text{HI}) \times P \rightarrow \mathbb{N}$*. If any instance of* τ_i *starts execution for the* $(n'_{ik}+1)$*th time on any core* π_k *at time* t*, then one of the following two reconfigurations are immediately performed on* π_k*:*

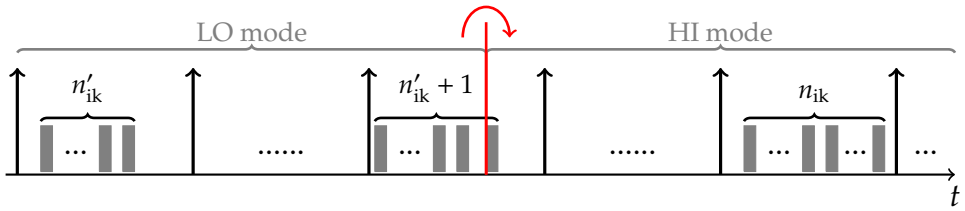
1- *All LO criticality tasks that are currently running at* t *or will arrive after* t *are dropped.*

2- *All future arrivals of LO criticality tasks are degraded with a service degradation factor* d_f*.*

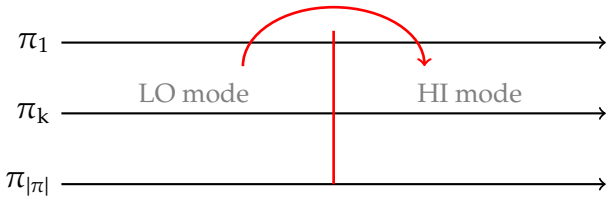
The service degradation factor d_{f} characterizes the new minimal inter-arrival time of any task $\tau_{i} \in \tau(\text{LO})$ after degradation (i.e., the minimal inter-arrival time T_{i} increases to $d_{f} \cdot T_{i}$ immediately after service degradation is triggered). We assume its value is provided by the system designer; our proposed analysis techniques in this work will further help to validate whether the provided degradation factor is feasible or not regarding both, system schedulability and safety. For notational convenience, we denote with $N'_{k}(\text{HI}) = \{n'_{ik} \mid \tau_{i} \in \tau^{k}(\text{HI})\}$ the adaptation profile of all HI criticality tasks on core π_{k}. With $N'_{k}(\text{HI})$, we define LO and HI modes on any core π_{k} as follows.

Definition 3.3 (Core Mode)**.** *Consider core* π_k *alone: If no instance of any* $\tau_i \in \tau^k(\text{HI})$ *executes for the* $\left(n'_{ik}+1\right)$*th time, then* π_k *is in LO mode. Otherwise,* π_k *is in HI mode, see Figure 3.1(a).*

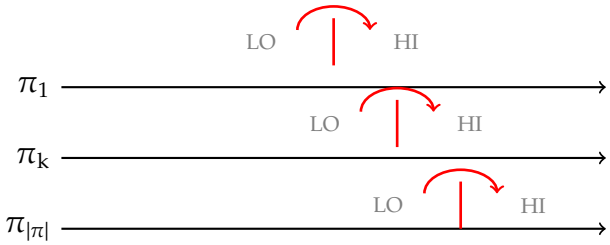
In the current work, we do not consider the switching from HI mode back to LO mode. This would result in more complex safety analysis,



(a) Core modes with redundancy & adaptation profiles - if a HI criticality task τ_i's instance exceeds on core π_k its LO mode re-execution profile (n'_{ik}, also called its adaptation profile), π_k enters HI mode where any HI criticality task τ_i's job can execute up to n_{ik} times (HI mode re-execution profile) and LO criticality tasks are dropped or degraded (Definition 3.2).



(b) Global switching



(c) Local switching

Figure 3.1: Core modes, global & local switching

which is out of the scope of this chapter.

It is still open how different cores coordinate their mode switches from LO to HI mode. There exists options to either trigger system reconfigurations synchronously on all cores (global switching) or asynchronously on each core (local switching). We formally define the options as follows.

Definition 3.4 (Global Switching). *All cores in our system switch from LO to HI mode (Definition 3.3) at the same time whenever mode switch is signaled on any core, see Figure 3.1(b).*

Definition 3.5 (Local Switching). *Any core in our system switches from LO to HI mode (Definition 3.3) independently of other cores, see Figure 3.1(c).*

We explore the trade-offs between global and local switching in this chapter: Global switching leads to a greatly reduced number of system states since at any time the platform can be in either HI or LO mode. This simplifies system safety analysis as compared to local switching, whose possible number of system states is $2^{|\pi|}$ at any time t (each core can be in either HI or LO mode). However, global switching is pessimistic as reconfiguration on any core can be triggered by any other core.

3.1.5 Fault-Tolerant Mixed-Criticality Scheduling

We have introduced above the redundancy and adaptation profiles to achieve a safe and efficient design of mixed-criticality systems, assuming transient errors. The remaining question is how to schedule a system given those profiles.

Our key observation is that the fault-tolerant mixed-criticality scheduling problem can be directly *transformed* into a classical mixed-criticality scheduling problem. We now briefly discuss the conventional mixed-criticality scheduling problem [BD16], we then establish a link between this problem and the fault-tolerant mixed-criticality scheduling problem in the current chapter.

Classical mixed-criticality scheduling problem: Analogous to Section 3.1.4, this problem assumes that the system starts with a nominal (LO) mode, where all tasks are guaranteed with their "normal" WCETs $(C_i(\text{LO}), \forall \tau_i)$. Exceeding normal WCETs is forbidden for LO criticality tasks but allowed for HI criticality tasks. Whenever any HI criticality task exceeds its LO mode WCET, the system enters HI mode. Thereafter, HI criticality tasks are allowed larger WCETs $(C_i(\text{HI}), \forall \tau_i \in \tau(\text{HI}))$, while LO criticality tasks are dropped or degraded. In this context, both partitioned [GSHT14] and global [HGA+15] scheduling techniques are proposed for mixed-criticality systems on multi-core platforms. Note that tasks of different criticality levels are *asymmetrically isolated* as LO criticality tasks cannot interfere with HI criticality tasks, while the opposite is allowed. This improves resource efficiency and differs from industrial approaches which enforce static temporal and spatial isolation [DO-92]. A detailed description of this problem has already been presented in Section 2.1.

Establish the link: For the problem in this work, first, since task instances are statically mapped to each core and task migration is forbidden due to excessive cost (see Definition 3.1), it follows that we have a classical partitioned scheduling problem. Second, on each core π_k, system reconfiguration is triggered when any HI criticality task τ_i's instance executes too many times, see Definition 3.2. We can translate this into a sufficient condition that, whenever any HI criticality task τ_i's

instance exceeds its LO WCET $C_{ik}(\text{LO}) = n'_{ik}C_i$ on core π_k, then system reconfiguration is triggered. Afterwards, core π_k is in HI mode, and we guarantee a HI criticality WCET for any HI criticality task τ_i, defined as $C_{ik}(\text{HI}) = n_{ik}C_i$.

For any instance of a LO criticality task τ_i, it cannot exceed its allocated number of redundancies; therefore, we have $C_{ik}(\text{HI}) = C_{ik}(\text{LO}) = n_{ik}C_i$ for all LO criticality tasks. We shall use $C_{ik}(\chi)$ to denote the WCET of τ_i on χ criticality level and on core π_k. In other words, the original mixed-criticality problem deals with timing errors (i.e., task overrun), thus it uses multiple WCETs for a task on different criticality levels. In this chapter, we focus on hardware/software transient errors, and task re-executions are modeled as overrun in the standard mixed-criticality model.

In summary, the impact of reconfigurations on schedulability is considered implicitly assuming classical mixed-criticality scheduling techniques. For the fault-tolerant mixed-criticality scheduling problem, a mixed-criticality task set $\Gamma(\tau, P, N(\cdot), N'(\cdot))$ can be constructed given τ, P, $N(\cdot)$ and $N'(\cdot)$, as discussed above. In this notation, any task can be duplicated into multiple sub-tasks running on different cores due to replication. Thus, we can adopt well-studied mixed-criticality scheduling techniques [BD16] to schedule tasks locally on each core. We focus in this work on the partitioned EDF-VD method [BCLS14], where EDF-VD [BBD+12] scheduling is adopted locally on each core. However, such an approach can be easily extended to consider other scheduling techniques. EDF-VD follows the standard mixed-criticality model and adopts EDF scheduling in both LO and HI modes. It uses shortened virtual deadlines (VD) for HI criticality tasks in LO mode, such that enough slack time is left between the virtual and actual deadlines to accommodate job overruns.

Finally, we define the problem studied in Chapter 3 as follows.

Definition 3.6 (Fault-Tolerant Mixed-Criticality Scheduling on Multi-cores (FMSM))**.** *Given a set of identical cores P, a dual-criticality sporadic task set τ and the probability of failure f_i for any instance of each task τ_i. Assume partitioned EDF-VD scheduling [BCLS14], find $N(\cdot)$ (Definition 3.1), $N'(\cdot)$ (Definition 3.2), such that both safety and schedulability of $\Gamma(\tau, P, N(\cdot), N'(\cdot))$ are satisfied with global switching (Definition 3.4) or local switching (Definition 3.5).*

The core of the above problem is the system safety analysis under $N(\cdot)$ and $N'(\cdot)$, while system schedulability is guaranteed with partitioned EDF-VD and under a problem transformation. We shall focus on safety analysis in this chapter and leverage standard search methods (see Section 3.4) to find $N(\cdot)$ and $N'(\cdot)$ once the safety analysis is performed.

3.2 Safety Analysis Under Global Switching

We quantify in this section system safety on different criticality levels assuming global switching (Definition 3.4), with or without task killing or service degradation.

3.2.1 Basic Safety Quantification

First, we study safety quantification without applying task killing or service degradation, i.e., basic safety quantification. For HI criticality tasks, this is apparently needed as they can never be killed or degraded in our model; for LO criticality tasks, basic safety quantification is also interesting as it will serve as a baseline to compare with the other cases when reconfigurations are applied. Recall that any instance of τ_i can execute at most n_{ik} times on core π_k and n_i times on all cores for one job arrival. We refer to n_i times of executions of one arrived instance of τ_i as *one round*. In the worst-case, a round of τ_i fails if all n_i instances of τ_i fail in this round. To further quantify the worst-case pfh in any interval, we need to upper bound the maximum number of rounds of any task τ_i in any interval. Formally, we have the following result.

Lemma 3.1. *Given τ, P and $N(\cdot)$, the maximum number of rounds of τ_i that any time interval of length t can accommodate, $r_i(N(\cdot), t)$, is given by:*

$$r_i(N(\cdot), t) = \max\left\{\left\lfloor (t + D_i - \max_k \{n_{ik}\} \cdot C_i) \middle/ T_i \right\rfloor, 0\right\} + 1. \tag{3.1}$$

Proof. This lemma is proved by an induction process as shown in Figure 3.2. The shortest time interval length that can accommodate one round failure is simply zero, supposing that the failure just falls in this interval (say, it happens at time instant t). In order to have a shortest time interval to accommodate one more round failure, we have to let the two failures happen as close as possible. The latest time instant the last round can start is $\left(t - \max_k\{n_{ik}\} \cdot C_i\right)$, hence the second last round starts at $\left(t - \max_k\{n_{ik}\} \cdot C_i - T_i\right)$ and fails at $\left(t' = t - \max_k\{n_{ik}\} \cdot C_i - T_i + D_i\right)$, which is the time instant that this round ends and the latest time instant it can fail. Similarly, the third last round fails at time instant $(t' - T_i)$. Therefore, the shortest time interval lengths that can accommodate one, two and three failures of τ_i are zero, $\left(\max_k\{n_{ik}\} \cdot C_i + T_i - D_i\right)$ and $\left(\max_k\{n_{ik}\} \cdot C_i + 2 \cdot T_i - D_i\right)$, respectively. By induction, the shortest length

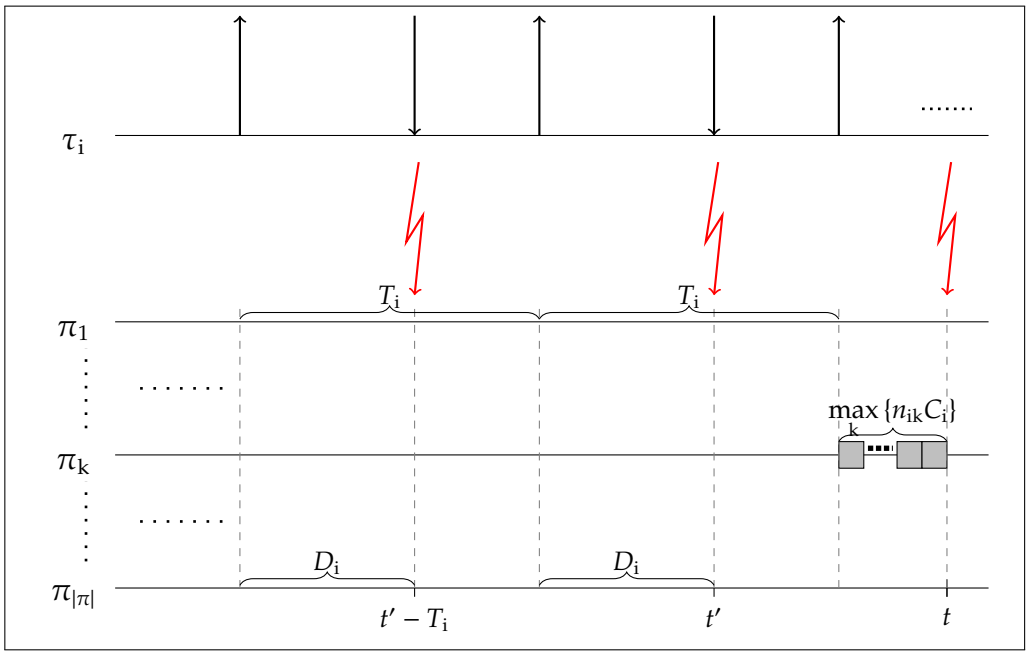


Figure 3.2: Induction process for Lemma 3.1

of the time interval that can accommodate n round failures of τ_i equals

$$\max\left\{\left(\max_k\{n_{ik}\}\cdot C_i + (n-1)\cdot T_i - D_i\right), 0\right\}. \tag{3.2}$$

Based on equation (3.2), the maximum number of rounds of τ_i in any time interval of length t is derived, as shown in equation (3.1). Note that we use floor ($\lfloor\ \rfloor$) here to calculate the value of $(n-1)$. □

Lemma 3.1 derives the worst-case number of rounds for any task in a time interval *on all cores*. Similarly, we can also upper-bound the local rounds of τ_i in any time interval on a core π_k, as shown in the following result.

Corollary 3.1. *Consider core π_k, given τ and $N(\cdot)$, the maximum number of local rounds of τ_i in any time interval of length t, $r_{ik}(n_{ik}, t)$, is given by:*

$$r_{ik}(n_{ik}, t) = \max\{\lfloor (t + D_i - n_{ik}\cdot C_i)/T_i \rfloor, 0\} + 1. \tag{3.3}$$

Proof. Proof can be similarly derived as in Lemma 3.1. □

With Lemma 3.1, we can upper-bound the pfh of any criticality level. The worst-case failure rate of any task τ_i in a given time interval happens when this interval accommodates the maximum number of rounds of τ_i

and each round fails; adding up those failure rates over all tasks of one critical level, we get the pfh on this criticality level. This is summarized into the following result.

Theorem 3.1. *Given τ, P and $N(\cdot)$, $pfh(\chi)$ (probability-of-failure-per-hour of χ criticality) can be upper bounded by:*

$$pfh(\chi) = \sum_{\tau_i \in \tau(\chi)} r_i(N(\cdot), t) \cdot f_i^{n_i}, \quad t = 1\ hour. \tag{3.4}$$

Proof. For any task $\tau_i \in \tau(\chi)$, the probability that τ_i does not fail in any time interval of length t is lower bounded by

$$(1 - f_i^{n_i})^{r_i(N(\cdot),t)}, \tag{3.5}$$

since all task instances fail/succeed independently. Thus, the probability that any $\tau_i \in \tau(\chi)$ fails in any time interval of length t, can be upper bounded by:

$$1 - \prod_{\tau_i \in \tau(\chi)} (1 - f_i^{n_i})^{r_i(N(\cdot),t)}. \tag{3.6}$$

Due to $f_i^{n_i} \ll 1$, we only consider its first order terms. Hence equation (3.6) can be tightly upper-bounded by:

$$1 - \left(1 - \sum_{\tau_i \in \tau(\chi)} r_i(N(\cdot), t) \cdot f_i^{n_i}\right) = \sum_{\tau_i \in \tau(\chi)} r_i(N(\cdot), t) \cdot f_i^{n_i}. \tag{3.7}$$

Thus, the upper bound of pfh (χ) can be computed by setting $t = 1\ hour$ as shown in equation (3.4). □

Summarizing, Lemma 3.1 and Theorem 3.1 enable us to quantify the upper bound of probability of failure per hour of any criticality level without system reconfigurations. Note that in Theorem 3.1, one could set t to O_s hours and derive the average pfh in the system operation. We do not consider this since for our assumption ($T_i \ll 1$ hour), equation (3.4) tightly upper-bounds the average pfh.

3.2.2 Safety Quantification with Task Killing

We proceed to quantify the system safety under task killing and global switching. As discussed, this only has an impact on the safety of LO criticality tasks. We will consider a system operation of O_s hours, after which the system state is reset. O_s will directly affect the probability of task killing and consequently the system safety.

Recall that on any core π_k, for any $\tau_i \in \tau^k(\text{HI})$, killing LO criticality tasks is controlled by n'_{ik}: if any instance of τ_i executes for the $\left(n'_{ik}+1\right)$th time, then all LO criticality tasks are killed immediately on all cores.

We first quantify the probability that killing LO criticality tasks is triggered by core π_k within a time interval of length t; we calculate this by using its complementary probability. Subsequently, we can derive the probability of task killing globally triggered by any core. Formally, we present the following lemma to quantify the probability that task killing or service reconfiguration is not triggered on one core.

Lemma 3.2. *Suppose that $\tau^k(\text{HI}) \neq \emptyset$ and $N'_k(\text{HI})$ is given. Then, the probability that no instance of any $\tau_i \in \tau^k(\text{HI})$ starts executing for the $(n'_{ik}+1)$th time on π_k in a time interval of length t is lower bounded by:*

$$R_k\left(N'_k(\text{HI}), t\right) = \prod_{\tau_i \in \tau^k(\text{HI})} (1 - f_i^{n'_{ik}})^{r_{ik}(n'_{ik}, t)}. \tag{3.8}$$

Proof. $\forall \tau_i \in \tau^k(\text{HI})$, the probability that an instance does not execute for the $(n'_{ik}+1)$th time in each round is $(1-f_i^{n'_{ik}})$, since this is the complementary probability that all the first n'_{ik} executions of τ_i's instance fail. The maximum number of rounds of τ_i on core π_k in a time interval of length t is $r_{ik}(n'_{ik}, t)$ according to equation (3.3). Thus, the probability that no instance of τ_i executes for the $(n'_{ik}+1)$th time in a time interval of length t is lower bounded by $(1 - f_i^{n'_{ik}})^{r_{ik}(n'_{ik}, t)}$. Equation (3.8) is the product of such probabilities over all tasks in $\tau^k(\text{HI})$, which lower-bounds the probability that no instance of any $\tau_i \in \tau^k(\text{HI})$ executes for the $(n'_{ik}+1)$th time in a time interval of length t on core π_k.

□

According to equation (3.8), $R_k(N'_k(\text{HI}), t)$ decreases with increasing t, which implies that the probability of task killing (or service reconfiguration) increases as time elapses – $R_k(N'_k(\text{HI}), t)$ approaches 0 if t is infinity, thus killing LO criticality tasks eventually occurs for certain. For HI criticality tasks, their safety is not affected as task killing is only performed for LO criticality tasks and cannot hamper the execution of HI criticality tasks. For LO criticality tasks, an instance of any $\tau_i \in \tau(\text{LO})$ does not fail if it is not killed and one of its replicas succeeds. We can thus first bound the failure probability of each round for any task. We then derive the upper bound of failure rates of any $\tau_i \in \tau(\text{LO})$ in a time interval by adding the failure probability of each round in this interval and by maximizing the number of rounds of τ_i in it. This is summarized in Theorem 3.2.

Theorem 3.2. *Given τ, P, $N(\cdot)$ and $N'(\cdot)$, assuming task killing and global switching, pfh(HI) can be calculated as shown in Theorem 3.1, since HI criticality tasks are not affected. Furthermore, for any $\tau_i \in \tau(\mathrm{LO})$, define $y_i(t)$ as a sequence of timing points unique to τ_i:*

$$y_i(t) = \{t - \max_k \{n_{ik}\} \cdot C_i - m \cdot T_i + D_i \mid m \in \mathbb{N} \wedge m \geq 1 \wedge m < r_i(N(\cdot), t)\} \cup \{t\}. \tag{3.9}$$

pfh(LO) can then be upper-bounded by:

$$\frac{1}{O_s} \sum_{\tau_i \in \tau(\mathrm{LO})} \sum_{\alpha \in y_i(t)} 1 - \left(\prod_{k=1}^{K} R_k(N'_k(\mathrm{HI}), \alpha) \right) \cdot \left(1 - f_i^{n_i}\right), \tag{3.10}$$

where $t = O_s$ denotes the system operation hours.

Proof. For one round of any $\tau_\mathrm{i} \in \tau(\mathrm{LO})$ finishing at time t, the probability that τ_i fails in this round can be upper bounded by:

$$1 - \left(\prod_{k=1}^{K} R_\mathrm{k}(N'_\mathrm{k}(\mathrm{HI}), t) \right) \cdot (1 - f_\mathrm{i}^{n_\mathrm{i}}). \tag{3.11}$$

Our explanations are as follows: In a time interval of length t, the probability that no instance of any $\tau_\mathrm{i} \in \tau^\mathrm{k}(\mathrm{HI})$ executes for the $(n'_\mathrm{ik} + 1)$th time on all cores can be lower bounded by:

$$\prod_{k=1}^{K} R_\mathrm{k}\left(N'_\mathrm{k}(\mathrm{HI}), t\right). \tag{3.12}$$

In addition, equation (3.12) also lower bounds the probability that LO criticality tasks are not killed under global switching. $(1 - f_\mathrm{i}^{n_\mathrm{i}})$ is the probability that τ_i does not fail by itself in this round. Thus, the product $\left(\prod_{k=1}^{K} R_\mathrm{k}(N'_\mathrm{k}(\mathrm{HI}), t)\right) \cdot (1 - f_\mathrm{i}^{n_\mathrm{i}})$ lower bounds the probability that τ_i does not fail in this round; its complement equation (3.11) computes the upper bound of the probability that τ_i fails in this round.

To get the worst-case failure rate for $\tau_\mathrm{i} \in \tau(\mathrm{LO})$ in a time interval of length t, we need to maximize the number of rounds in it, simply because this leads to the maximum number of rounds that can fail. Since equation (3.8) decreases with increasing t, each round should finish as late as possible to maximize failure probability, see equation (3.11). Applying Lemma 3.1, the $r_\mathrm{i}(N(\cdot), t)$th round, in the latest case, starts at $(t - \max_\mathrm{k}(n_\mathrm{ik}) \cdot C_\mathrm{i})$ and ends at t; similarly, the $(r_\mathrm{i}(N(\cdot), t) - 1)$th round

starts at $(t - \max_{k}(n_{ik}) \cdot C_i - T_i)$ and finishes at $(t - \max_{k}(n_{ik}) \cdot C_i - T_i + D_i)$. By induction, in the worst case, the $(r_i(N(\cdot), t) - m)$th round starts at $(t - \max_{k}(n_{ik}) \cdot C_i - m \cdot T_i)$ and must finish by $(t - \max_{k}(n_{ik}) \cdot C_i - m \cdot T_i + D_i)$ (except for the last round, which finishes at t). Thus, for any $\tau_i \in \tau(\mathrm{LO})$, we use $y_i(t)$ to denote this sequence of latest finishing times for all rounds of τ_i, while the total number of rounds is maximum.

With the above results, we can upper bound the probability of failure of the round of τ_i finishing at $\alpha \in y_i(t)$ by $1 - \left(\prod_{k=1}^{K} R_k(N'_k(\mathrm{HI}), \alpha\right) \cdot (1 - f_i^{n_i})$. Therefore, pfh (LO) can be upper bounded by equation (3.10) (upper bound of failure rate during system operation of O_s hours divided by O_s) under global switching and task killing. □

Theorem 3.2 states that the safety of HI criticality tasks are not affected under task killing since those tasks can always execute based on their redundancy profiles. Theorem 3.2 further upper-bounds the pfh of LO criticality level under global switching and task killing. This is done by first identifying a worst-case scenario $y_i(t)$ for each task τ_i and then applying it to every LO criticality task. Our result confirms that the safety of LO criticality tasks depends on the adaptation profiles of HI criticality tasks according to equation (3.10). With decreasing adaptation profiles, LO criticality tasks are killed more often, leading to reduced system safety if killed tasks are safety related. One can also observe that as the system operation duration increases, the pfh of LO criticality level also tends to be worse, since LO criticality tasks are more likely to be killed as time progresses.

3.2.3 Service Degradation instead of Task Killing

Though task killing can greatly improve system schedulability [BD16], in practice, service degradation is often preferred due to constant service requirements on all criticality levels. In addition, task killing might directly violate system safety, as LO criticality tasks (could still be safety-related) are completely removed in such a drastic approach.

When service degradation is triggered, LO criticality tasks will have a service degradation factor d_f larger than one – for any $\tau_i \in \tau(\mathrm{LO})$, its minimal inter-arrival time becomes $d_f \cdot T_i$. In addition, service degradation on all cores are synchronously triggered due to global switching. We now quantify the impact of service degradation on system safety under global switching – for a system operation interval of O_s hours, find the worst-case switching time, such that the total failure rates of any task in this interval are maximized (worst-case). This is summarized in Theorem 3.3.

Theorem 3.3. *Given τ, P, $N(\cdot)$ and $N'(\cdot)$, assuming service degradation under global switching, pfh (HI) can be upper-bounded by Theorem 3.1 as HI criticality tasks are not affected. Additionally, we define $s_i(N(\cdot), d_f, t)$ as the maximum number of rounds that $\tau_i \in \tau(\mathrm{LO})$ can fail (or accommodate) in an interval of length t with service degradation factor d_f if service degradation is triggered at the beginning of this time interval:*

$$s_i(N(\cdot), d_f, t) = \max\left\{\left\lfloor \frac{t + D_i - \max_k \{n_{ik}\} \cdot C_i}{d_f \cdot T_i} \right\rfloor, 0\right\} + 1. \tag{3.13}$$

Furthermore, $z\left(d_f, t\right)$ is defined as the upper bound of failure probability of LO criticality tasks in this time interval:

$$z\left(d_f, t\right) = \sum_{\tau_i \in \tau(\mathrm{LO})} s_i(N(\cdot), d_f, t) \cdot f_i^{n_i}. \tag{3.14}$$

pfh (LO) can then be upper bounded by

$$\frac{1}{O_s}\left(1 - \prod_{k=1}^{K} R_k(N'_k(\mathrm{HI}), t)\right) \cdot z\left(1, t\right), \tag{3.15}$$

where $t = O_s$ hours.

Proof. First of all, suppose that service of any $\tau_\mathrm{i} \in \tau(\mathrm{LO})$ is degraded with a service degradation factor d_f in a time interval of length t on all cores. Equation (3.13) can be derived according to Lemma 3.1. In addition, according to Theorem 3.1, equation (3.14) upper-bounds the probability of failure of LO criticality tasks in a time interval of length t if service degradation of LO criticality tasks is already triggered. We assume global switching happens at time instant λt ($0 \leq \lambda \leq 1$). Based on equation (3.12), the probability that LO criticality tasks are not degraded till λt under global switching is lower bounded by:

$$\prod_{k=1}^{K} R_\mathrm{k}\left(N'_\mathrm{k}(\mathrm{HI}), \lambda t\right). \tag{3.16}$$

The complement of equation (3.16) upper bounds the probability that LO criticality tasks are degraded at time instant λt. For a system operation interval of length t, from starting to λt, service degradation of LO criticality tasks on all cores is not triggered thus probability of failure of LO criticality tasks is upper bounded by $z\left(1, \lambda t\right)$. From λt to t, service of LO criticality tasks is degraded with d_f on all cores; hence, failure probability of LO criticality tasks is upper bounded by $z\left(d_\mathrm{f}, (1-\lambda)t\right)$.

Therefore, probability of failure of LO criticality tasks on all cores in a system operation interval of length t under global switching and service degradation can be upper bounded by:

$$\left(1 - \prod_{k=1}^{K} (R_{\mathrm{k}}(N'_{\mathrm{k}}(\mathrm{HI}), \lambda t)\right) (z\,(1, \lambda t) + z\,(d_{\mathrm{f}}, (1-\lambda)t))\,. \tag{3.17}$$

Based on the proof of Lemma 3.2, equation (3.16) decreases while λ increases. Furthermore, with increasing λ, $z\,(1, \lambda t)$ increases while $z\,(d_{\mathrm{f}}, (1-\lambda)t)$ decreases. However, the increasing rate of $z\,(1, \lambda t)$ is larger than the decreasing rate of $z\,(d_{\mathrm{f}}, (1-\lambda)t)$. This is because for a time interval of fixed length, if service degradation of $\tau(\mathrm{LO})$ is triggered, any $\tau_{\mathrm{i}} \in \tau(\mathrm{LO})$ accommodates less number of failures than it can accommodate without service degradation. Thus, $(z\,(1, \lambda t) + z\,(d_{\mathrm{f}}, (1-\lambda)t))$, accordingly the upper bound of pfh (LO), is maximum when $\lambda = 1$. □

Theorem 3.3 calculates the pfh of LO criticality level under service degradation and global switching. This is done by finding the worst-case mode switching time within a system operation of O_s hours, such that the total failure rates of LO criticality tasks are maximized. The choice of O_s depends on the system under consideration; for our analysis, it is required to bound the mode switch probability and to search for a worst-case scenario. Together with Theorem 3.2, this concludes our safety analysis under global switching.

3.3 Safety Analysis Under Local Switching

We proceed to perform safety analysis under local switching (Definition 3.5). Unlike global switching as presented in the previous section, local switching provides the flexibility for each core to switch from LO to HI mode independently, reducing the pessimism that LO criticality tasks must be killed or degraded on all cores at once. However, this complicates the analysis as the possible system states explode (i.e., when and which cores incur reconfigurations). We focus on analyzing the case with task killing. An analysis for service degradation under local switching is provided in [ZHT16].

3.3.1 Task Killing Under Local Switching

Our key intuition is that, despite the combinatorial nature of the problem, we can still perform a round per round analysis to pessimistically bound system failure rates. The main reason is that, for any LO criticality task

τ_i on any core π_k, we can bound the probability of executing each round. This enables us to further bound the failure probability of each round on each core independently of other cores.

Recall that any instance of $\tau_i \in \tau(\text{LO})$ can execute at most n_{ik} times on core π_k for one job arrival. A round of τ_i fails on π_k if it is killed or all n_{ik} instances of τ_i fail in this round on π_k. We quantify the upper bound of failure probability of one round of τ_i finishing at time t on each core π_k individually. The product of such probability across all cores bounds the failure probability of one round of τ_i finishing at time t. Furthermore, by taking the maximum number of rounds τ_i can encounter in a time interval and adding up the failure probability of each round, we get the worst-case failure rate of a single task; from this, pfh(LO) can be further derived. This analysis is formally presented in Theorem 3.4.

Theorem 3.4. *Given τ, P, $N(\cdot)$ and $N'(\cdot)$, assuming task killing under local switching, pfh (HI) can be upper-bounded by Theorem 3.1 as HI criticality tasks are not affected. We further define $u_{ik}(t)$ as the upper bound of failure probability of one round of $\tau_i \in \tau^k(\text{LO})$ finishing at time t on π_k:*

- *if $n_{ik} = 0$,*

$$u_{ik}(t) = 0, \tag{3.18}$$

- *if $\exists \tau_j \in \tau^k(\text{HI}) : n_{jk} \neq 0 \wedge n'_{jk} = 0$,*

$$u_{ik}(t) = 1, \tag{3.19}$$

- *if $\left((N_k(\text{HI}) = \emptyset) \vee \left(N_k(\text{HI}) = N'_k(\text{HI})\right)\right) \wedge (n_{ik} \neq 0)$,*

$$u_{ik}(t) = f_i^{n_{ik}}, \tag{3.20}$$

- *if $\left(N'_k(\text{HI}) \neq \emptyset\right) \wedge \left(N_k(\text{HI}) \neq N'_k(\text{HI})\right) \wedge (n_{ik} \neq 0)$,*

$$u_{ik}(t) = 1 - R_k\left(N'_k(\text{HI}), t\right) \cdot \left(1 - f_i^{n_{ik}}\right). \tag{3.21}$$

Using $y_i(t)$ (see equation (3.9)), pfh(LO) can be upper bounded by:

$$\frac{1}{O_s}\left(\sum_{\tau_i \in \tau(\text{LO})} \sum_{\alpha \in y_i(t)} \left(\prod_{\substack{k=1 \\ u_{ik}(\alpha) \neq 0}}^{K} u_{ik}(\alpha)\right)\right), \tag{3.22}$$

where $t = O_s$ hours.

Proof. Recall that, for any core π_k, killing $\tau_i \in \tau(\text{LO})$ is adopted when any

$\tau_j \in \tau^k(\text{HI})$ executes for the $\left(n'_{jk}+1\right)$th time; we derive the upper bound of failure probability of one round of τ_i finishing at t on π_k as $u_{ik}(t)$ in four cases.

1. If $n_{ik} = 0$, then there is no instance of τ_i executing on π_k. Hence, τ_i does not fail on π_k, we have equation (3.18).

2. If $\exists \tau_j \in \tau^k(\text{HI}) : n_{jk} \neq 0 \wedge n'_{jk} = 0$, then τ_j executes on π_k with "zero" adaptation profile, meaning that any $\tau_i \in \tau^k(\text{LO})$ will be killed when the first instance of τ_j starts to execute. Thus, in the worst case, any $\tau_i \in \tau(\text{LO})$ will not execute (i.e., simply fail), we have equation (3.19).

3. If $\left((N_k(\text{HI}) = \emptyset) \vee \left(N_k(\text{HI}) = N'_k(\text{HI})\right)\right) \wedge (n_{ik} \neq 0)$, then on π_k, there are no HI criticality tasks or redundancy profile of HI criticality tasks equals their adaptation profile; in both cases, killing LO criticality tasks will not be triggered on π_k. Hence, one round of $\tau_i \in \tau^k(\text{LO})$ fails with probability $f_i^{n_{ik}}$ on π_k, we have equation (3.20).

4. If $\left(N'_k(\text{HI}) \neq \emptyset\right) \wedge \left(N_k(\text{HI}) \neq N'_k(\text{HI})\right) \wedge (n_{ik} \neq 0)$, then killing LO criticality tasks is triggered on π_k when any task $\tau_j \in \tau^k(\text{HI})$ executes the $(n'_{jk}+1)$th time. Based on our system model, for one round of $\tau_i \in \tau^k(\text{LO})$ ends at t, $R_k(N'_k(\text{HI}), t)$ lower bounds the probability that it is not killed until t. Therefore, the probability that it does not fail is lower bounded by $R_k(N'_k(\text{HI}), t) \cdot \left(1 - f_i^{n_{ik}}\right)$, whose complement yields the upper bound of the probability that it fails, see equation (3.21).

Second, to get the worst-case failure rate for $\tau_i \in \tau(\text{LO})$ in a time interval of length t, we need to get the maximum number of rounds it can accommodate in this time interval so that the number of rounds it can fail is maximum. In addition, all rounds on all cores should end as late as possible to get the maximum value of equation (3.21). According to the proof of Theorem 3.2, for any $\tau_i \in \tau(\text{LO})$, $y_i(t)$ is a sequence of the latest time that each round of τ_i finishes while there are maximum number of rounds in the considered interval.

As shown above, equations (3.18), (3.19), (3.20) and (3.21) upper bound $u_{ik}(t)$ for any $\tau_i \in \tau(\text{LO})$ in all possible cases. Thus, based on our system model, one round of $\tau_i \in \tau(\text{LO})$ finishing at time t fails if it fails on all cores with probability upper bounded by:

$$\prod_{\substack{k=1 \\ u_{ik}(t) \neq 0}}^{K} u_{ik}(t) . \tag{3.23}$$

The complement of equation (3.23),

$$1 - \prod_{\substack{k=1 \\ u_{\mathrm{ik}}(t)\neq 0}}^{K} u_{\mathrm{ik}}(t), \tag{3.24}$$

lower-bounds the probability that one round of $\tau_{\mathrm{i}} \in \tau(\mathrm{LO})$ finishing at t does not fail. Thus, the probability that $\tau_{\mathrm{i}} \in \tau(\mathrm{LO})$ does not fail in a time interval of length t can be lower bounded by the probability that $r_{\mathrm{i}}(N(\cdot), t)$ rounds have all executed successfully:

$$\prod_{\alpha \in y_{\mathrm{i}}(t)} \left(1 - \prod_{\substack{k=1 \\ u_{\mathrm{ik}}(\alpha)\neq 0}}^{K} u_{\mathrm{ik}}(\alpha) \right). \tag{3.25}$$

Taking the complement of equation (3.25), the probability of failure of $r_{\mathrm{i}}(N(\cdot), t)$ rounds of $\tau_{\mathrm{i}} \in \tau(\mathrm{LO})$ in a time interval of length t can be upper bounded by

$$1 - \prod_{\alpha \in y_{\mathrm{i}}(t)} \left(1 - \prod_{\substack{k=1 \\ u_{\mathrm{ik}}(\alpha)\neq 0}}^{K} u_{\mathrm{ik}}(\alpha) \right). \tag{3.26}$$

Due to $\prod_{\substack{k=1 \\ u_{\mathrm{ik}}(\alpha)\neq 0}}^{K} u_{\mathrm{ik}}(\alpha) \ll 1$, we only consider its first order terms, equation (3.26) can be tightly upper-bounded by

$$\sum_{\alpha \in y_{\mathrm{i}}(t)} \left(\prod_{\substack{k=1 \\ u_{\mathrm{ik}}(t)\neq 0}}^{K} u_{\mathrm{ik}}(t) \right). \tag{3.27}$$

Therefore, the upper bound of pfh(LO) in a system operation of O_{s} hours can be calculated as the average overall failure rates in this interval, as shown in equation (3.22). □

Theorem 3.4 upper-bounds the pfh of LO criticality level under task killing and local switching. From the analysis, we can observe that as the system operates longer (i.e., O_s increases), the pfh of LO criticality also increases as LO criticality tasks are more likely to be killed.

3.4 Evaluation

We validate in this section our proposed techniques with both a real-life flight management system (FMS) application and synthetic task sets. As

for the FMS application, we show the impact of task killing and service degradation on system safety and schedulability; with synthetic task sets, we present the improvements of system feasibility achieved by task killing or service degradation.

3.4.1 Flight Management System

Task killing or service degradation under global/local switching are evaluated with a real-life flight management system, and their impact on system safety and schedulability is studied. The considered FMS consists of 7 DO-178B criticality level B tasks and 4 criticality level C tasks, with detailed parameters found in Table 2.2. As we consider HI = B and LO = C, the safety requirements are pfh(HI) $< 10^{-7}$ and pfh(LO) $< 10^{-5}$. We assume any instance of each task is subject to a constant failure probability 10^{-5} [HYT14a]. We conduct our experiments on dual-core/quad-core platforms: For all cases, we first find a feasible redundancy profile $N(\cdot)$ to satisfy system safety and schedulability without system reconfigurations, while load balancing is achieved at the same time by applying a genetic optimization algorithm [ga]. We then fix $N(\cdot)$ and increase the adaptation profile $N'(\cdot)$ of all HI criticality tasks to show the impact (assuming a uniform adaptation profile for all HI criticality tasks). The service degradation factor d_f is set to 5 in case of degrading LO criticality tasks. We present our results in Figures 3.3 and 3.4, where the x-axis represents the values of the adaptation profile of all HI criticality tasks, the left y-axis (uMax) represents the maximum utilization factor across all cores and the right y-axis represents the upper bound of pfh (LO) in logarithmic scale. In our experiments, the upper bound of pfh (HI) always satisfies safety requirements. pfh (HI) in logarithmic scale equals -35.17 and -75.17 on dual-core and quad-core systems, respectively. Additionally, the impact of the adoption of task killing or service degradation is evaluated.

3.4.1.1 Analysis of Experiment Results

First of all, we can observe that in all cases, with increasing values of adaptation profile of all HI criticality tasks, system schedulability is hampered while safety is enhanced. With a larger adaptation profile, LO criticality tasks are killed/degraded less likely (often), see our analysis in Section 3.2 and Section 3.3. This will not help for system schedulability, as we would kill or degrade LO criticality tasks frequently to improve schedulability. However, system safety can be improved, simply because LO criticality tasks are killed/degraded less likely. For example, with task killing under global switching on a dual-core platform (Figure 3.3(a)), the maximum utilization across all cores is increased from 0.345 to 0.603

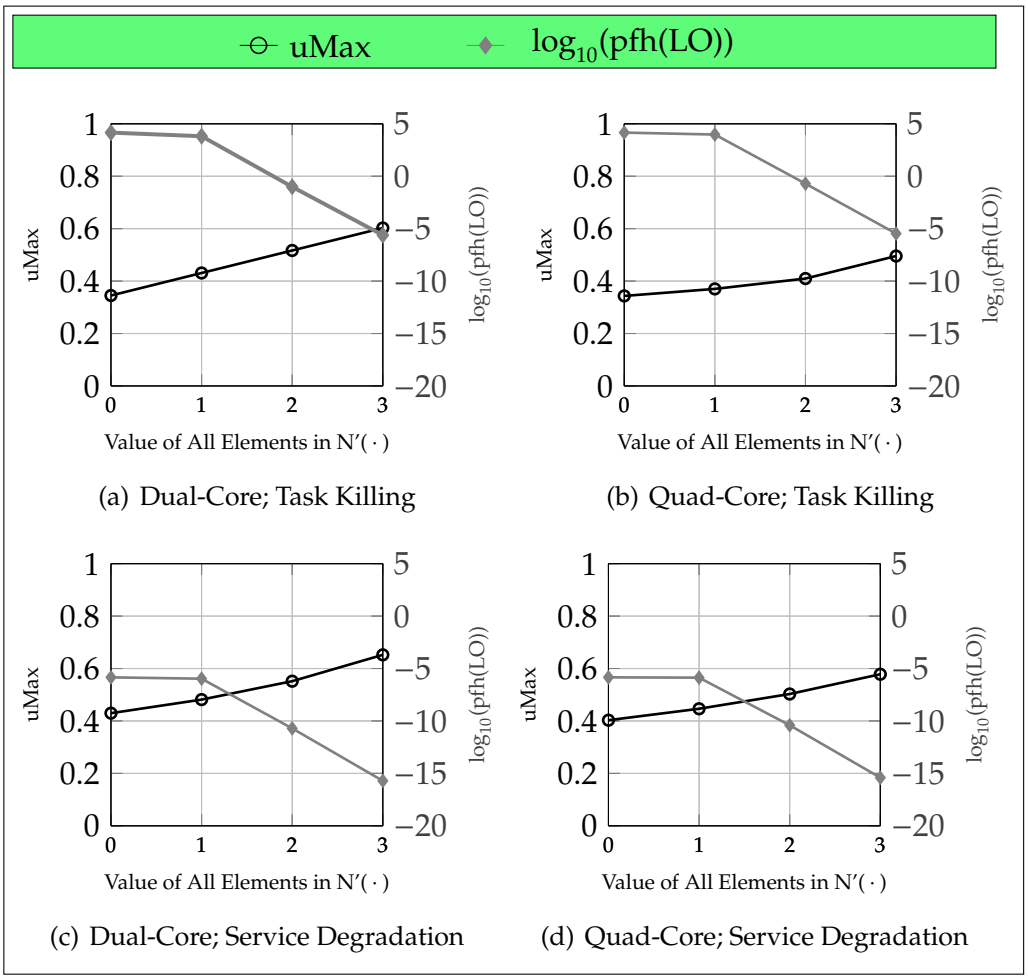


Figure 3.3: FMS: Global Switching

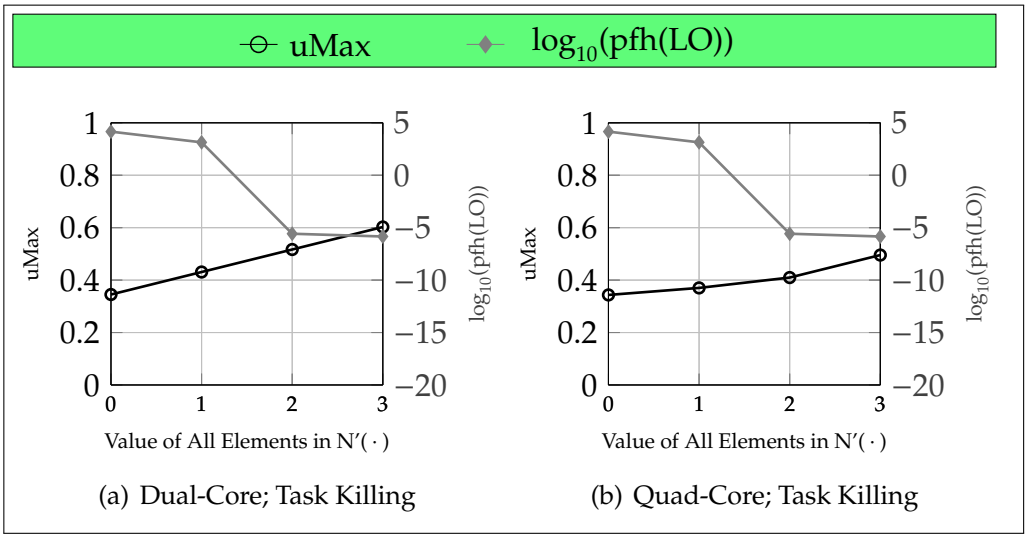


Figure 3.4: FMS: Local switching

with adaptation profile increasing from 0 to 3; at the same time, pfh(LO) decreases by 10 orders of magnitude.

Next, as expected, we can observe that with more cores, system schedulability can be improved. For instance, considering an adaptation profile of 3 under service degradation and global switching (Figure 3.3(c) and 3.3(d)), the maximum utilization factor across all cores on dual-core (0.652) is larger than that (0.578) on quad-core. However, the difference here is not large: In order to evaluate the impact of adaptation profile on system feasibility, we fixed the redundancy profile of all 7 HI criticality tasks on each core to 4. Thus, load balancing is only performed for 4 LO criticality tasks of low utilizations, which will not result in a large difference in the maximum utilization across all cores.

Furthermore, if we compare task killing with service degradation under global switching (e.g., Figure 3.3(a) and Figure 3.3(c)), we can observe that task killing improves system schedulability more, while jeopardizing more system safety. Task killing is more abrupt than service degradation; when triggered, it removes *all* LO criticality tasks either locally (local switching) or globally (global switching), freeing more resources to guarantee schedulability of HI criticality tasks while affecting the safety of LO criticality tasks. For example, as shown in Figure 3.3, task killing satisfies system safety only with an adaptation profile of 3, while service degradation satisfies safety requirements in all cases.

Last, by comparing global with local switching (e.g., task killing in Figure 3.3(b) and Figure 3.4(b)), we can observe that both incur similar impacts on system schedulability, since the worst-case is that all LO criticality tasks are killed to guarantee schedulability of HI criticality tasks. However, global switching has a stronger impact on system safety – in global switching, killing LO criticality tasks can be triggered by any core, leading to a significantly higher killing probability for LO criticality tasks than in case of local switching. Regarding task killing in all cases (Figure 3.3(a), 3.3(b), 3.4(a) and 3.4(b)), system safety is guaranteed with a larger adaptation profile under global switching ($\forall \tau_i \in \tau(\text{HI}), \forall \pi_k \in P : n'_{ik} = 3$) than that under local switching ($\forall \tau_i \in \tau(\text{HI}), \forall \pi_k \in P : n'_{ik} = 2$).

3.4.1.2 Summary of Results

We conclude that system safety and schedulability contradicts with each other under system reconfiguration. To improve system safety, either the redundancy or the adaptation profile needs to be increased. However, this increases system utilization and hampers schedulability. Those trade-offs can be quantified by the analysis techniques proposed in this chapter.

3.4.2 Synthetic Task Sets

We proceed to validate the analysis and design methods on synthetic task sets. It is well known that task killing or service degradation can improve mixed-criticality system schedulability [BD16]. With explicit safety analysis under system reconfigurations, we study here whether they can help improve system feasibility if safety is also involved. To this end, we adopt a well-known random mixed-criticality task generator to generate 200 task sets at each system utilization point [BD16]. The task generation is slightly modified as compared to [BD16] to fit our system model in Section 3.1. We repetitively apply random search to each task set to find feasible design parameters under various assumptions (e.g., local/global switching, task killing/service degradation); if the search process is successful then the task set is said to be feasible (otherwise not). We then compare the system acceptance ratios (the ratio of the number feasible task sets to the number of tested task sets) for those different assumptions at each data point.

3.4.2.1 Task Generation

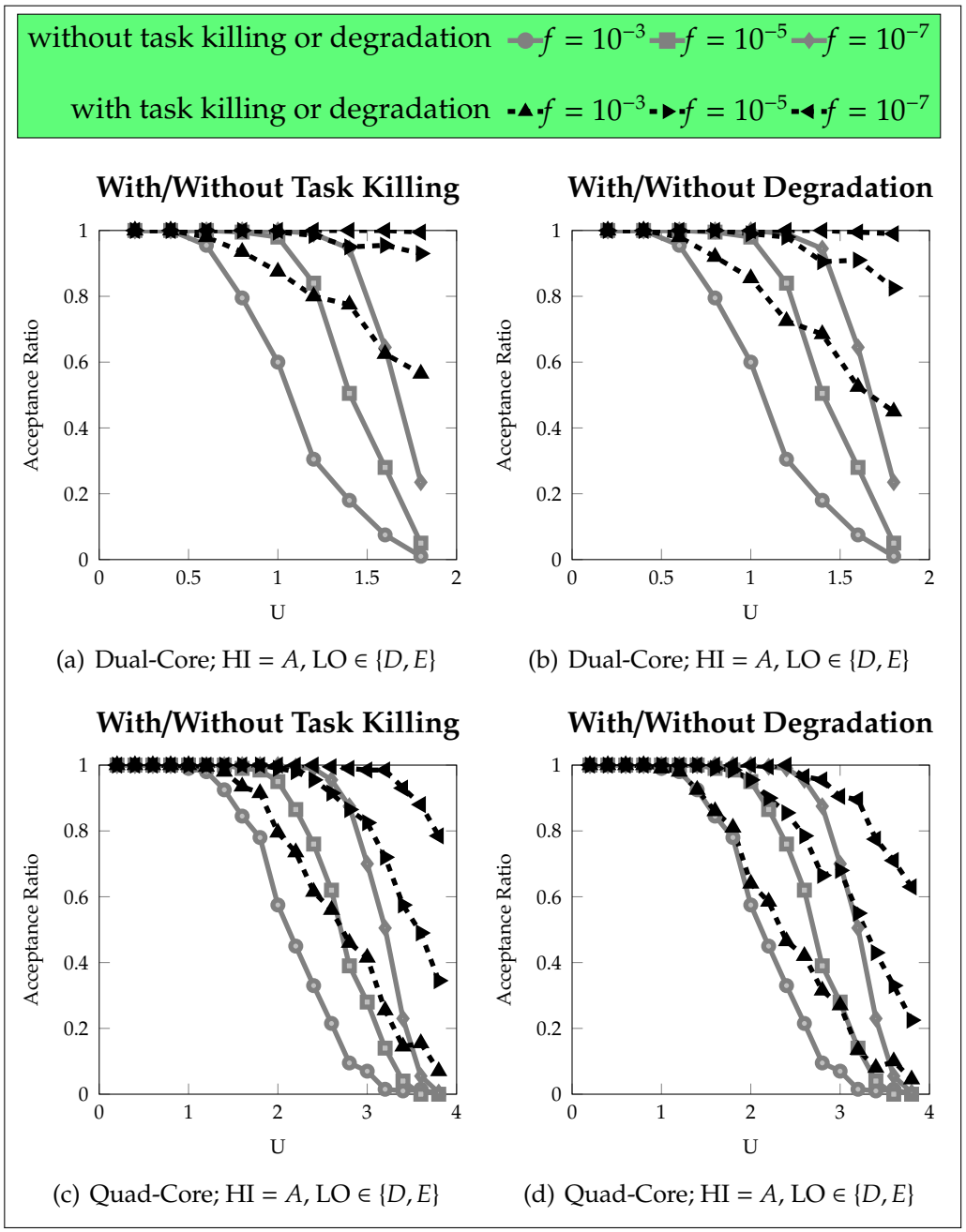
Random implicit deadline dual-criticality task sets are generated. We adopt a similar task generator to that shown in Section 2.3.3.1, with the following configuration of controlling parameters:

- The utilization ($\frac{C_i}{T_i}$) of any task τ_i is uniformly drawn from $[0.02, 0.2]$.
- The system utilization U is defined as $\sum_{\tau_i} \frac{C_i}{T_i}$.
- The minimal inter-arrival time of each task is generated uniformly from $[2, 2000]$ms.
- The probability a generated task has HI criticality P_{HI} is set to 0.2.

The random task generator starts with system utilization factor $U = 0.2$ and randomly adds new tasks into this task set until a certain U is reached (U in increments of 0.2). Notice that, system reconfiguration is applied only if it is not feasible without reconfiguration.

3.4.2.2 Analysis of Results

We present our results in Figures 3.5, 3.6 and 3.7, where the x-axis represents system utilization before applying redundancies or adaptations, the y-axis represents the acceptance ratio, and f represents



(a) Dual-Core; HI = A, LO $\in \{D, E\}$

(b) Dual-Core; HI = A, LO $\in \{D, E\}$

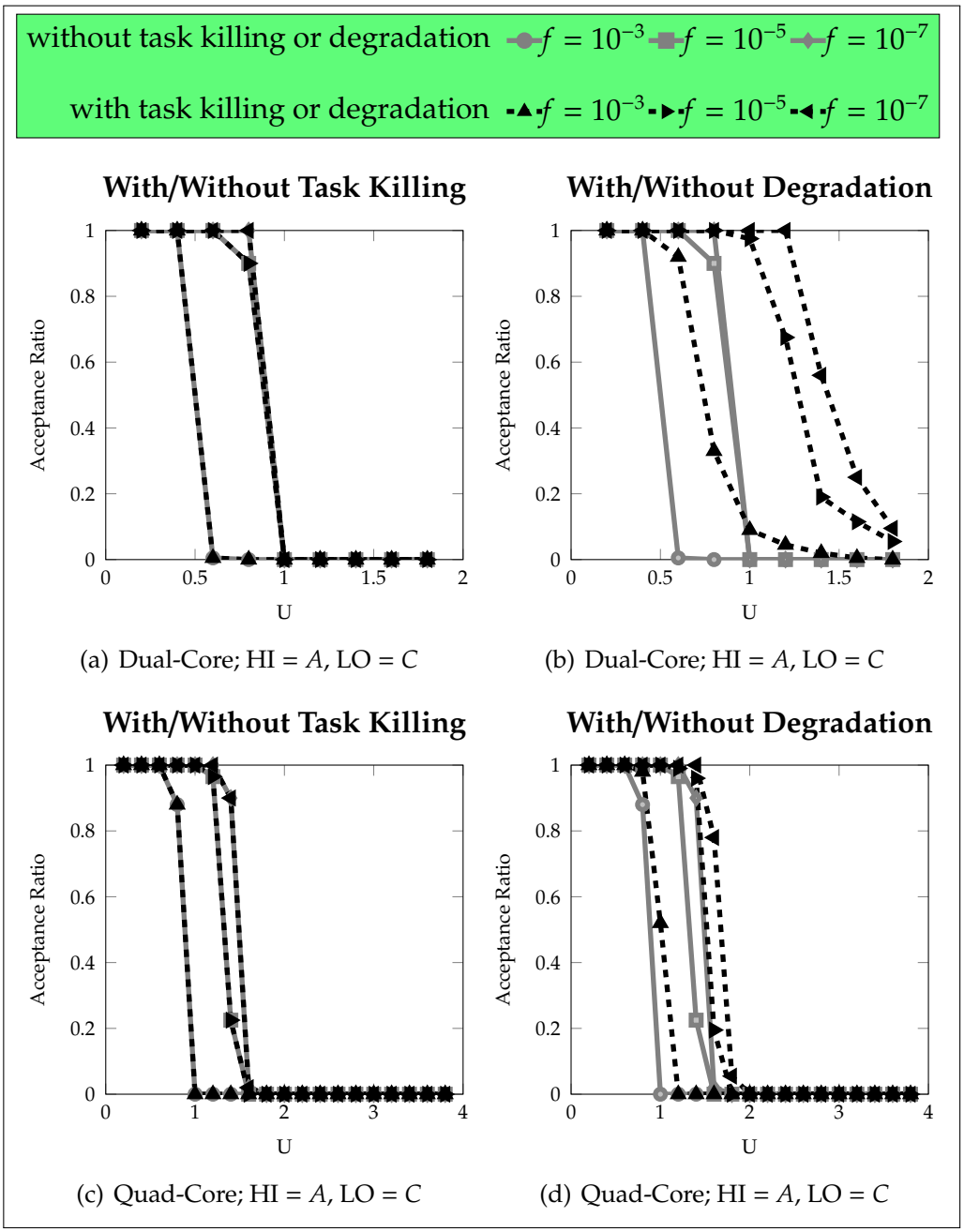
(c) Quad-Core; HI = A, LO $\in \{D, E\}$

(d) Quad-Core; HI = A, LO $\in \{D, E\}$

Figure 3.5: Feasibility evaluation with global switching

the universal failure probability of any task instance. We provide here our results for dual-core and quad-core systems.

First, as expected, it can be observed that system feasibility constantly improves with more reliable hardware platforms (decreasing f) or more cores. For example, consider system utilization $U = 1$ on a dual-core/quad-core platform (LO = E) without task killing. The acceptance ratio increases from 60% to 100% as f decreases from 10^{-3} to 10^{-7} on a



(a) Dual-Core; HI = A, LO = C

(b) Dual-Core; HI = A, LO = C

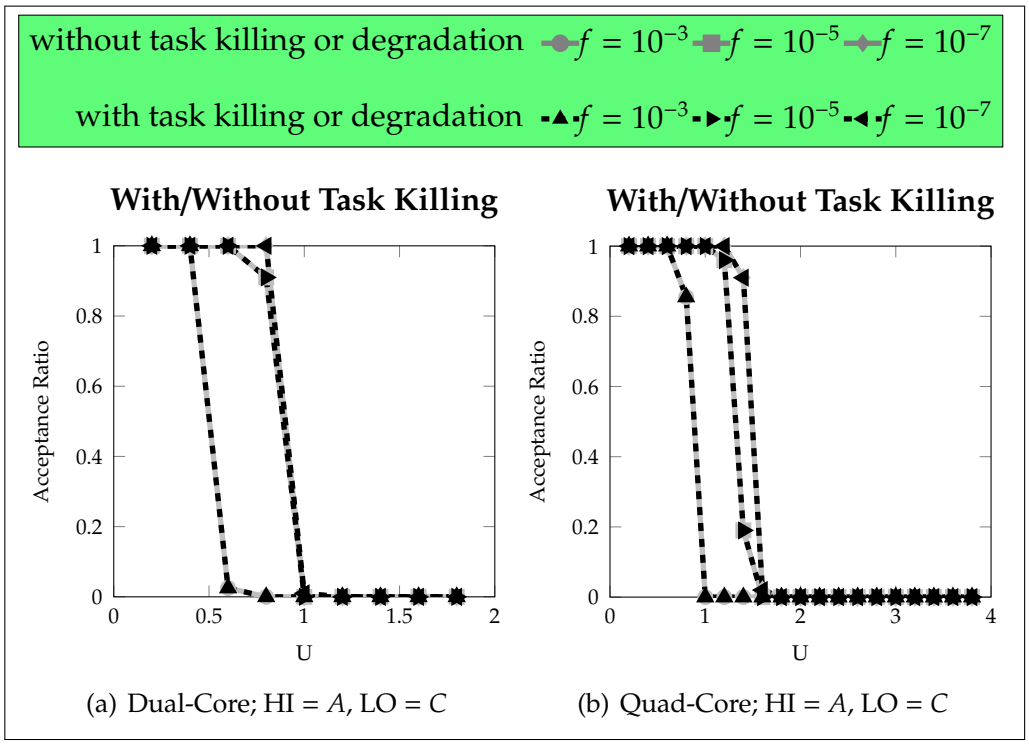
(c) Quad-Core; HI = A, LO = C

(d) Quad-Core; HI = A, LO = C

Figure 3.6: Feasibility evaluation with global switching

dual-core platform (Figure 3.5(a)). With $f = 10^{-3}$, the acceptance ratio increases from 60% (Figure 3.5(a)) to 99% (Figure 3.5(c)) as the number of cores increases from 2 to 4.

Next, according to the results shown in Figure 3.5, if LO criticality tasks are not safety related (LO = E), system feasibility can be improved by both task killing and service degradation. In addition, task killing performs better than service degradation here, since it can safely remove



(a) Dual-Core; HI = A, LO = C (b) Quad-Core; HI = A, LO = C

Figure 3.7: Feasibility evaluation with local switching

all LO criticality tasks to improve schedulability. For example, consider $U = 2$ and $f = 10^{-3}$ on a quad-core platform (Figure 3.5(c) and 3.5(d)). Acceptance ratio increases by 38.26%/11.30% under killing/degradation as compared to the case when only redundancies are applied.

Moreover, as shown in Figure 3.6, if LO criticality tasks are safety related (LO = C), then global switching with task killing hardly helps the design of fault-tolerant mixed-criticality systems because task killing can immediately violate system safety. In detail, the reason is two-fold: First, after killing, those safety-related LO criticality tasks constantly fail. Second, as time elapses, the probability that killing is triggered constantly increases, eventually reaching 1 (see Lemma 3.2). However, global switching with service degradation improves system feasibility and it is more proper when LO criticality tasks are safety concerned – as long as the system accepts degraded services, it is always beneficial for safety as less task instances are executed. Thus, LO criticality tasks can only fail less. For instance, consider $U = 1$ and $f = 10^{-3}$ on a dual-core platform (Figure 3.6(a) and 3.6(b)), acceptance ratio does not improve under task killing, whereas it increases by 9% under service degradation.

Last, similar to global switching, for local switching (Figure 3.7), killing LO criticality tasks when they are safety-related could rarely help system feasibility, as this could immediately violate system safety.

3.4.2.3 Summary of Results

We conclude that task killing greatly improves system feasibility if LO criticality tasks are not safety related. However, if safety is a concern for LO criticality tasks, service degradation is a proper choice as task killing could directly violate system safety. Again, the analysis methods derived in this chapter enable us to quantify those findings.

3.5 Summary

Fault-tolerant mixed-criticality scheduling under hardware/software transient errors on multi-core (with single-core as a special case) was studied in this chapter. We proposed the first framework known to date, where runtime adaptation, mixed safety requirements and task deadlines are considered jointly in the design of mixed-criticality systems.

Specifically, we developed techniques that could reconfigure the system at run-time such that critical tasks can still be guaranteed under urgent situations, i.e., when they do not succeed after a certain number of trials. With explicit modeling of system safety on different criticality levels according to well-known safety standards [DO-92], we can quantify the impact of online reconfigurations (task killing or service degradation) on safety. To guarantee schedulability, we performed a problem transformation and reduced the mixed-criticality fault-tolerant scheduling problem to a classical mixed-criticality scheduling problem. Finally, with both a flight management system and synthetic task sets, we validated our proposed techniques and demonstrated quantitatively two important findings: Task killing as commonly assumed for mixed-criticality systems can hardly help improving system feasibility if it is performed on safety-related functions; except for this case, system feasibility can be improved to a great extent under run-time system reconfigurations.

4

Energy-Aware Mixed-Criticality – Concepts, Methods and Implications

Energy minimization is a prime requirement in the design of (embedded) computer systems, not only due to cost reduction reasons, but also due to the related thermal issues [Kum14]. As an example, a modern car can have up to 100 electronic control units (ECUs) [Wik16b]. This requires a tremendous power supply. In order to sustain the system computing performance, energy minimization in the cyber space becomes vital. Furthermore, with aggressive shrinking of circuit footprint, power density of modern electronic circuits is drastically increasing. This may result in overheating and lead to system failures if not handled properly [EBA+11, Won08].

Challenges: Although there is a rich body of literature on energy minimization techniques for conventional real-time safety-critical systems [CK07, BBDM00], applying them to mixed-criticality systems might not be straightforward. On one hand, well-established methods like Dynamic Voltage and Frequency Scaling (DVFS) [YWA+11] or Dynamic Power Management (DPM) [BBPDM99] can be applied to mixed-criticality systems. On the other hand, mixed-criticality systems can react to runtime threats (mainly task overrun) by switching between different operation modes and making different service guarantees on different criticality levels. This could consequently complicate the reduction of energy due to the following challenges.

1 **Unclear energy objective:** For mixed-criticality systems, there is a bounded uncertain workload we have to guarantee for critical tasks (for the sake of system safety, see [BD16]). Though assumed to be improbable, the uncertain workload complicates energy minimization as we do not know for which workload we are going to minimize system energy. Hence, to conserve energy for mixed-criticality systems, a proper energy objective needs to be justified in the first place.

2 **Conflict between energy minimization and system safety:** The main idea of applying DVFS to minimize energy consumption is to stretch task execution times as much as possible by lowering the processor frequency, such that tasks finish "just in time". In other words, DVFS tries to explore all slack in the system and pushes the execution of tasks to the time limit. On the contrary, in order to guarantee system safety, we have to reserve time budget for critical tasks, such that they can still meet their deadlines even if they overrun. This needs to be prepared a priori and prevents us from exploring all available slack in the system. Such a conflict needs to be taken into account when solving the mixed-criticality energy minimization problem.

3 **Energy efficient task to processor mapping:** Mixed-Criticality systems advocate to consolidate functionalities of different criticality levels in the same commercial-off-the-shelf (COTS) computing platform. Consequently, this could lead to compact systems with reduced costs. However, if energy reduction is the primary goal, mixing tasks of different criticality levels might not be beneficial as current research on mixed-criticality systems [BD16] tends to pack the system workload onto a minimal set of processors, hampering load balancing and energy conservation. Addressing this requires new energy efficient task mapping techniques to be developed.

Contribution: To tackle the above challenges, we present in this chapter solutions on applying DVFS to mixed-criticality systems in order to save energy. Different from Section 2.4 where we apply DVFS to speedup the system and to guarantee less critical tasks in urgent scenarios, we apply DVFS in this chapter to slowdown the system in order to save energy. Due to the hardness of the underlying mixed-criticality scheduling problem, we focus on integrating energy saving techniques into a well-known mixed-criticality scheduling algorithm – EDF-VD [BBD+12]. Furthermore, we minimize both static and dynamic energy consumption while exploring the trade-off between different system operation modes. We achieve this by first proposing an optimal

solution and an effective lightweight heuristic on a single-core. We then extend these results to multi-core by designing new energy-aware task mapping techniques. Our detailed contributions are as follows.

- We clarify the characteristic of the mixed-criticality energy minimization problem and show that a fundamental trade-off exists between energy consumptions in different system modes: Speeding up to handle overrun increases system energy consumption in urgent scenarios but reduces the energy consumption in nominal scenarios, and vise versa.
- We integrate DVFS with the EDF-VD scheduling technique and formulate the single-core energy minimization problem as a convex program. We further develop more theoretical insights into this problem and present a lightweight heuristic solution achieving near-optimal results.
- We extend our single-core solutions to multi-core by first extending existing mixed-criticality task mapping techniques and making them energy-aware. Furthermore, we develop a drastically different task mapping technique by spatially isolating tasks of different criticality levels on different cores while performing load balancing. The latter strategy is appealing for reasons that are 2-fold: (1) Avoiding mixing tasks of different criticality levels on the same processing core removes mutual interferences among them; this leads to tighter schedulability tests and more system slack can be exploited by DVFS to save energy. (2) Strong criticality isolation is favored in practice due to its ease of certification [EN16].
- We demonstrate with experimental results energy saving and various trade-offs for the mixed-criticality energy minimization problem. Our results reveal an energy saving up to 36% on synthetic task sets and 40% for a flight management system. We also demonstrate that energy saving under spatial isolation among different criticality levels is comparable to the case when tasks of different criticality levels are mixed on each core.

Organization: This chapter is organized as follows: We introduce in Section 4.1 the necessary background on the underlying system model. In Section 4.2 we present an example to motivate our problem formulation. We propose an optimal analytical solution to our energy minimization problem on a single-core in Section 4.3, while in Section 4.4 we present a corresponding simple, yet effective, heuristic solution. Leveraging our single-core solutions, we present different energy-aware task mapping techniques in Section 4.5. We evaluate and conclude our proposed techniques in Section 4.6 and Section 4.7.

Related Work: The state-of-the-art research on mixed-criticality systems has primarily focused on providing heterogeneous timing guarantees for tasks of different criticality levels. To date, only a few research results have been published to tackle the challenging problem of energy minimization for mixed-criticality systems [LJP13, Guo14, VHL14]. Those publications, although making different assumptions for their specific problems, all share a common concept to sacrifice the performance of low criticality tasks in order to manage or optimize system energy: In [LJP13], an approach based on DPM is presented to trade deadlines of low criticality tasks for energy saving on multi-core, while deadlines of high criticality tasks are always guaranteed. A DVFS method minimizing energy for single-core mixed-criticality systems is presented in [Guo14], while respecting system reliability requirements. Rather than treating energy as an optimization goal, energy utilization on critical tasks is advocated in [VHL14] when the system is short of energy supply. To our best knowledge, our proposed mixed-criticality energy minimization solutions [HKGT14, NHG+16] are the first ones designed under the state-of-the-art mixed-criticality task model [BD16].

4.1 System Model

In this chapter, we assume the widely adopted dual-criticality task model: A system consists of a set of sporadic tasks with implicit deadlines, and each task is of either HI or LO criticality. We focus on the EDF-VD scheduling technique [BBD+12] and its multi-core extension (partitioned EDF-VD [Bar14]). Details about the task model and the assumed scheduling techniques can be found in Section 2.1. We introduce in the following our assumed power model. For the convenience of the readers, all important notations in this chapter are summarized in Table 4.1.

4.1.1 Power Model and DVFS

We adopt a popular power model from [CK07, PC13a]. Assuming a homogeneous multi-core platform $\pi = \{\pi_1, \pi_2, ..., \pi_{|\pi|}\}$, the power consumption of any processor is formulated as

$$P(f) = P_s + \beta f^{\alpha}, \tag{4.1}$$

where $P(f)$ is the total power consumed and P_s stands for the static power consumption due to leakage current. f denotes the operating frequency of the processor. βf^{α} represents the dynamic power consumption caused by switching activities, where α and β are circuit dependent parameters. A

Table 4.1: Important notations in Chapter 4

f_b	frequency at which WCETs are estimated
$[f_{\min}, f_{\max}]$	available frequency range with DVFS
f_i^{χ}	frequency of task τ_i in χ mode
$f_{\chi 1}^{\chi 2}$	frequency of all χ_1 criticality tasks in χ_2 mode
$f_{\chi 1\ \text{opt}}^{\chi 2}$	optimal $f_{\chi 1}^{\chi 2}$ for single-core energy minimization
E_{χ}	system energy consumption in χ mode
w_{χ}	weight factor for χ mode
x	deadline shortening factor in EDF-VD
x_{LB}, x_{UB}	lower and upper bounds of x
$\hat{x}_{\text{LB}}$, $\hat{x}_{\text{UB}}$	lower and upper bounds of x at $f_i^{\chi} = f_{\max}$
x_{LBopt}, x_{UBopt}	optimal lower and upper bounds of x

common assumption is that $\alpha \geq 2$ [NMM+11, PC13b]. Hence, decreasing processor frequency leads to reduced dynamic energy consumption, as explored by the well-known DVFS technique [CK07]. However, with reduced frequency, leakage energy will increase as it takes longer for jobs to complete. Thus there is a critical frequency f_{crit} below which it is not beneficial to reduce frequency energy-wise. For any job with workload of nc clock cycles, [PC13b] shows that f_{crit} can be obtained as follows:

$$\frac{d(\frac{nc}{f}P_s + \frac{nc}{f}\beta f^{\alpha})}{df} = 0 \Leftrightarrow f_{\text{crit}} = \sqrt[\alpha]{\frac{P_s}{\beta(\alpha-1)}}. \tag{4.2}$$

As a result, we assume in this chapter that each core in a hardware platform is independently DVFS-capable and can execute with any frequency in $[f_{\min}, f_{\max}]$, where $f_{\min} \geq f_{\text{crit}}$. The WCETs of tasks are estimated at frequency f_b, where $f_{\min} \leq f_b \leq f_{\max}$. Notice that applying DVFS changes the actual WCETs of tasks, such that χ criticality WCET of task τ_i becomes $\frac{C_i(\chi)f_b}{f}$ while running at frequency f.

We focus on CPU processing energy in this chapter and neglect energy consumed by communication and memory sub-systems. According to [DFG+11, Vog10], CPU processing is the major energy source for modern computer systems, which can contribute up to 75%-80% of the total system power consumption. Under realistic but more complex models of task execution time and system energy [YWA+11], our proposed techniques could be extended to minimize additionally communication energy; this could be achieved by treating the communication phases of tasks as separate artificial tasks, as similarly done in [YWA+11].

Finally, we say that a processor is in active state if it is processing tasks; otherwise, the processor is in inactive state. We assume that the processor

energy consumption in the inactive state and the overhead of switching between active and inactive states are negligible. This assumption holds for platforms with deep-sleep modes and negligible power mode transition overhead (e.g., current NXP Kinetis series feature ~4μs wakeup time from the state-retaining deep-sleep mode [kin]). However, we believe that important findings of this work would still hold, e.g., energy trade-offs between different system modes and the benefit of isolated task mapping, see Section 4.3 and Section 4.5.

4.2 Problem Formulation

We motivate our work using a general setting for mixed-criticality energy minimization, where both static and dynamic power consumptions are considered in all system modes. Based on this, we provide a concrete problem formulation in this section.

4.2.1 Motivation

Current research on mixed-criticality systems advocates the mode-switched paradigm – a dual-criticality system starts with the nominal scenario or LO mode, where all tasks are guaranteed; if any HI criticality task overruns, then the system switches immediately to an urgent scenario or HI mode, where LO tasks are dropped.

The question then is to clarify the system energy objective while taking into account the different system modes. We may follow two approaches.

1 If we assume that the urgent scenario is unlikely, then a meaningful goal is to just optimize the nominal mode system energy.

2 If we take into account that the probability of switching from LO to HI mode increases with time (eventually reaching 1 [HYT14b]), then one should jointly consider energy consumption in all system modes. We can define the weight factor w_χ for χ mode ($\chi \in \{\text{HI}, \text{LO}\}$), which indicates the relative importance of minimizing system energy in χ mode. Such weight factors would help to make global energy saving decisions and to trade-off energy consumptions in different system modes. We assume such weight factors are normalized, i.e., $w_{\text{LO}} + w_{\text{HI}} = 1$.

In practice, one could interpret the weight factors as the percentages of time (or alternatively, the probabilities) that the system operates in the corresponding modes. It is important to note that the first approach is

merely a special case of the second one, i.e. $w_{\text{LO}} = 1 \wedge w_{\text{HI}} = 0$. Due to this reason, we will take the second approach.

The motivation to consider static energy is rather straightforward – static power could overwhelm dynamic power as the CMOS technology improves [KAB+03]. Thus, it is critical to minimize static energy along with dynamic energy [DVI+02].

We use the following motivational example to quantify the impact of weight factors and DVFS on energy minimization. We first evaluate the case when the system runs on a single-core and then proceed to multi-core platforms with an additional consideration of task mapping strategies.

Example 4.1. ***Single-Core:*** *Consider the task set as shown in Table 4.2, which is schedulable on a single-core under EDF-VD (base frequency $f_b = 2GHz$). The processor is DVFS-capable, where $[f_{\min}, f_{\max}] = [1, 2]GHz$. We assume $\alpha = 3$, $\beta = 1W/GHz^{\alpha}$ and $P_s = 1W$ [PC13b]. We calculate the total weighted energy consumption in* LO *and* HI *modes with varying weight factors w_{LO} (w_{HI}). The optimization is performed by the standard Mathematica optimization engine [mat]. We consider 2 strategies to show the impact of weight factors and DVFS on energy minimization:*

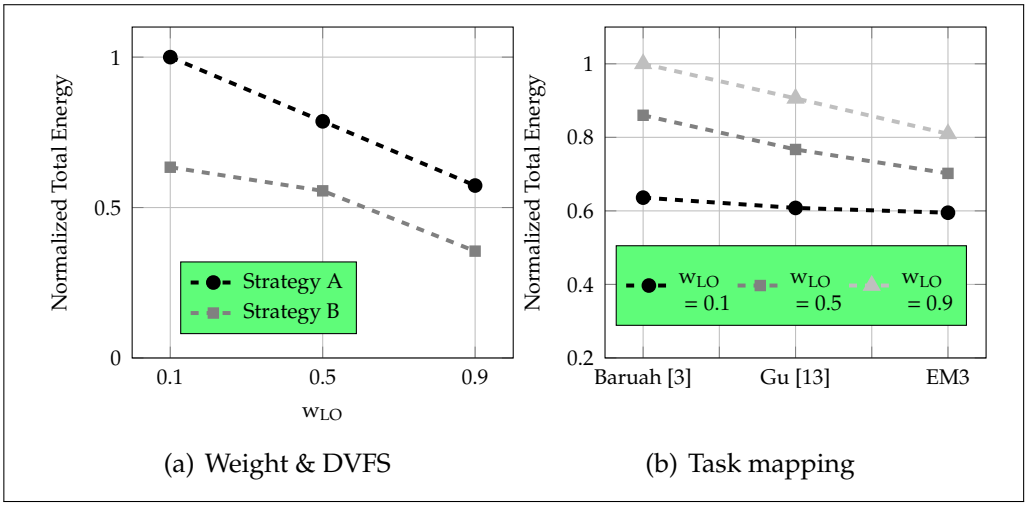
- ***SA*** *No DVFS is applied; all tasks execute on base frequency f_b.*
- ***SB*** *DVFS is applied to minimize both static and dynamic energy according to the problem formulation in Theorem 4.1 and the optimality criteria in Section 4.3.*

We summarize our results in Figure 4.1(a). We observe that energy can be greatly reduced by employing DVFS. If we compare the two strategies, 37% of total energy consumption is saved by employing DVFS when $w_{\text{LO}} = 0.1$. In addition, we observe that weight factors greatly affect the total energy consumption for both strategies. For example, in Strategy B, the minimal system energy consumption can vary by 44% when w_{LO} changes from 0.1 to 0.9. This further implies that the choice of weight factors is critical. That is, either neglecting HI *mode energy ($w_{\text{LO}} \to 1$) or neglecting* LO *mode energy ($w_{\text{LO}} \to 0$) would lead to overly optimistic or pessimistic results. However, detailed domain knowledge about the designed applications may be required to select meaningful weight factors.*

Multi-Core: *For our experiments here and for the purpose of a better illustration, we consider the Flight Management System (FMS) as shown in Table 4.3. We assume $f_{\min}/f_b/f_{\max} = 0.6/1.2/1.2GHz$, $\beta = 2W/GHz^{\alpha}$, $P_s = 0.8W$ and $\alpha = 3$. On multi-core platforms, we have to further consider the impact of task mapping on energy minimization. To this end, we compare three different approaches – two state-of-the-art mapping techniques (Baruah's method [Bar14] and Gu's method [GGDY14]) and a new method EM3 proposed in this work (see Section 4.5).*

Table 4.2: Task set for Figure 4.1(a) in Example 4.1 (task parameters in ms)

τ	χ_i	T_i	C_i(LO)	C_i(HI)
τ_1	HI	40	4	12
τ_2	HI	75	6	18
τ_3	HI	40	3	9
τ_4	LO	100	6	6
τ_5	LO	80	5	5



(a) Weight & DVFS (b) Task mapping

Figure 4.1: Motivational example – energy consumption under the impact of weight factors, DVFS and task mapping

We apply the aforementioned techniques to the FMS application on a dual-core platform. We first minimize the system energy consumption individually on each core. Then, the total energy for each mapping method is calculated by deriving the weighted energy consumption in different system modes on each core and then summing them up across all cores. Figure 4.1(b) shows the normalized total energy consumption obtained using different mapping techniques. As we can observe, indeed task mapping can greatly affect the achievable energy saving. As an example, with $w_{\mathrm{LO}} = 0.5$*, EM3 achieves up to* 19% *energy reduction compared to the previous methods. The reason for this is that our method can balance the workload better across different cores in order to improve energy saving. We provide detailed explanations in Section 4.5. Similar to the single-core case, we can also see that weight factors play a critical role in energy minimization for multi-core platforms. In summary, through the above example, we have demonstrated that, weight factors and task mapping both play a critical role in the mixed-criticality energy minimization problem. Motivated by this, we proceed to formally define the problem studied in this chapter.*

4.2.2 Problem Formulation

We divide-and-conquer the energy minimization problem by solving two sub-problems: (i) We first perform energy-aware task-to-processor mapping; (ii) we then develop single-core DVFS techniques and apply them to all cores. However, the first problem is dependent on the second, as we want to find mapping techniques that could explore best the energy saving potential of single-core solutions. Therefore, we will first study the sing-core problem and then proceed to solve the multi-core problem. In the following, we formulate these two subproblems.

4.2.2.1 Single-Core Problem

For this problem, we assume that τ is our considered task set on one core. To apply DVFS, the essential problem is to assign each task an operating frequency in each system mode, such that energy is minimized while mixed-criticality real-time guarantees are satisfied.

Let us denote the frequency of task τ_i in mode χ as f_i^{χ}, where $\chi \in \{\text{LO}, \text{HI}\}$. We can uniformly represent this frequency assignment using function $\mathbb{F} : \tau \times \{\text{LO}, \text{HI}\} \rightarrow \mathbb{R}^+$. To consider energy minimization for both system modes, we first need to define a proper energy objective. To achieve this, we express the importance of minimizing LO mode energy with a weight factor w_{LO} (similarly w_{HI} for HI mode). We assume that $w_{\text{LO}}, w_{\text{HI}} \in [0, 1]$ and $w_{\text{LO}} = 1 - w_{\text{HI}}$. We do not pose any restriction on how to obtain w_{LO} and w_{HI}. Our formulation is rather general: If $w_{\text{LO}} = 1 \wedge w_{\text{HI}} = 0$, we minimize only LO mode energy; if $w_{\text{LO}} = 0.5 \wedge w_{\text{HI}} = 0.5$, we then minimize the average energy consumption in both modes.

With the weight factors, we can formalize our energy objective. First, for one hyper-period in LO mode with length $\Pi_\tau T_i$, we calculate the normalized total energy consumption in this interval as the actual energy consumption divided by the interval length:

$$E_{\text{LO}} = w_{\text{LO}} \sum_{\tau_i \in \tau} \frac{C_i(\text{LO})}{T_i} \frac{f_b}{f_i^{\text{LO}}} (P_s + \beta (f_i^{\text{LO}})^\alpha). \tag{4.3}$$

Similarly, we normalize the system energy consumption in a HI mode hyper-period with length $\Pi_{\chi_i=\text{HI}} T_i$ as

$$E_{\text{HI}} = w_{\text{HI}} \sum_{\tau_i \in \tau_{\text{HI}}} \frac{C_i(\text{HI})}{T_i} \frac{f_b}{f_i^{\text{HI}}} (P_s + \beta (f_i^{\text{HI}})^\alpha). \tag{4.4}$$

Finally, our goal is to minimize the system energy E across both operation modes, where

$$E = E_{\text{LO}} + E_{\text{HI}}, \tag{4.5}$$

and the optimal task frequencies in different system modes need to be chosen (f_i^χ, $\forall\tau_i\forall\chi$). Furthermore, energy should be minimized while satisfying the mixed-criticality real-time requirements. Using Theorem 2.1, we can get a new schedulability condition when task utilizations are scaled under DVFS. We show the application of DVFS to EDF-VD, extending Theorem 2.1 using the following corollary.

Corollary 4.1. *Consider a task set τ scheduled by EDF-VD on a single-core. Assume that task τ_i has frequency f_i^χ in system mode χ under DVFS, then the feasible range of the deadline shortening factor x for* HI *criticality tasks is*

$$0 < \frac{\widetilde{U}_{\mathrm{HI}}^{\mathrm{LO}}}{1-\widetilde{U}_{\mathrm{LO}}^{\mathrm{LO}}} \leq x \leq [\![\frac{1-\widetilde{U}_{\mathrm{HI}}^{\mathrm{HI}}}{\widetilde{U}_{\mathrm{LO}}^{\mathrm{LO}}}]\!]^1, \tag{4.6}$$

where

$$\widetilde{U}_{\chi_1}^{\chi_2} = \sum_{\tau_i\in\tau_{\chi_1}} \frac{\widetilde{C}_i(\chi_2)}{T_i}; \quad \widetilde{C}_i(\mathrm{LO}) = \frac{C_i(\mathrm{LO})f_b}{f_i^{\mathrm{LO}}}, \forall\tau_i\in\tau;$$

$$\widetilde{C}_i(\mathrm{HI}) = \frac{C_i(\mathrm{LO})f_b}{f_i^{\mathrm{LO}}} + \frac{(C_i(\mathrm{HI})-C_i(\mathrm{LO}))f_b}{f_i^{\mathrm{HI}}}, \forall\tau_i\in\tau_{\mathrm{HI}}.$$

According to Corollary 4.1, with increasing f_i^χ for any task τ_i, the lower bound of x after DVFS does not increase, while the upper bound does not decrease as all utilization factors decrease. Thus, we know for any possible DVFS strategy, x must be within the absolute respective bounds when $f_i^\chi = f_{\max}$, $\forall\tau_i\forall\chi$. We denote the lower and upper bounds in this case as $\hat{x}_{\mathrm{LB}}$, $\hat{x}_{\mathrm{UB}} \in [0, 1]$, respectively.

To apply DVFS to save energy, we need to ensure that the energy objective (4.5) is minimized while condition (4.6) is satisfied. With reformatting, we can formulate our single-core energy minimization as a convex program. We capture this with the following theorem.

Theorem 4.1. *The single-core energy minimization problem can be formulated as a convex program given by:*

$$\textbf{\textit{minimize}} \quad E = E_{\mathrm{LO}} + E_{\mathrm{HI}}, \tag{4.7}$$

$$\textbf{\textit{s.t.}} \quad \frac{\widetilde{U}_{\mathrm{HI}}^{\mathrm{LO}}}{x} + \widetilde{U}_{\mathrm{LO}}^{\mathrm{LO}} \leq 1; \tag{4.8}$$

$$x\widetilde{U}_{\mathrm{LO}}^{\mathrm{LO}} + \widetilde{U}_{\mathrm{HI}}^{\mathrm{HI}} \leq 1; \tag{4.9}$$

$$x \in [\hat{x}_{\mathrm{LB}}, \hat{x}_{\mathrm{UB}}]; \tag{4.10}$$

$$\forall\tau_i\forall\chi, f_i^\chi \in [f_{\min}, f_{\max}]. \tag{4.11}$$

The convexity of the above program can be easily verified [Boy04]. Our energy objective (4.7) is a convex function of task frequencies, see the calculations of E_{LO} and E_{HI} in equations (4.3) and (4.4). In addition, the left-hand sides of constraints (4.8) and (4.9) are also convex functions of task frequencies and the deadline scaling factor. Although the convex formulation suggests practical algorithms [Boy04] to solve this problem, we will theoretically investigate the problem further and develop more insights in later sections.

4.2.2.2 Multi-Core Problem

Assuming partitioned scheduling in this work, we still need to find an energy efficient task mapping onto a multi-core platform, such that by applying single-core DVFS locally on each core, the total energy consumed by all tasks on all cores is minimized. Let us denote task mapping with function $\mathbb{M} : \tau \rightarrow \pi$. In addition, since we apply EDF-VD on each core locally, we define deadline scaling factors on all cores through function $\mathbb{X} : \pi \rightarrow \mathbb{R}^+$. Let us further denote with $E(k)$ the total energy consumption on core π_k, as calculated in equation (4.7). The multi-core energy minimization problem can then be formalized as follows.

Definition 4.1 (Mixed-Criticality Energy Minimization on Multi-Core)**.** *Given a task set τ on a DVFS-capable platform π, find $\mathbb{M}$, $\mathbb{X}$ and $\mathbb{F}$ such that the total energy consumption on all cores $\sum_{1 \leq k \leq |\pi|} E(k)$ is minimized.*

4.3 A Single-Core Optimal Solution

We proceed to acquire a deeper insight into the single-core energy minimization problem. Since we have a convex formulation (Theorem 4.1), we can first apply the Karush-Kuhn-Tucker (KKT) optimality conditions [KT51] to this problem. Due to the intrinsic computation complexity involved, we then show how to reduce the actual search space for single-core energy minimization.

4.3.1 KKT Conditions

Let us first introduce the KKT conditions for solving general optimization problems. Consider a problem of the form

$$\begin{aligned} \textbf{minimize} \quad & f(z), \\ \textbf{s.t.} \quad & h_i(z) = 0, \qquad \forall i = 1, ..., n; \\ & g_j(z) \leq 0, \qquad \forall j = 1, ..., m. \end{aligned}$$

where $h_i(z)$ and $g_j(z)$ are the equality and inequality constraints for a continuous function $f(z)$, respectively.

Theorem 4.2 (KKT conditions)**.** *According to [KT51], if the objective and constraint functions are continuously differentiable, then a local minimal solution z^* exists, when there exist Lagrange multipliers λ_i $(1 \leq i \leq n)$ and KKT multipliers μ_j $(1 \leq j \leq m)$, such that*

$$\nabla f(z^*) + \sum_{i=1}^{n} \lambda_i \nabla h_i(z^*) + \sum_{j=1}^{m} \mu_j \nabla g_j(z^*) = 0,$$

$$\textbf{\textit{s.t.}}\ h_i(z^*) = 0, \quad \forall i = 1, ...n; \quad g_j(z^*) \leq 0, \quad \forall j = 1, ...m;$$

$$\mu_j g_j(z^*) = 0, \quad \forall j = 1, ...m; \ \mu_j \geq 0, \quad \forall j = 1, ...m.$$

4.3.2 Problem Complexity When Applying KKT

As we have a convex program with continuous decision variables for our single-core problem (i.e., task frequencies are continuously available and the deadline scaling factor is also a continuous variable, see Theorem 4.1), we can directly apply Theorem 4.2 to find the optimal solution. Due to the convexity of the problem formulation, any local optima is guaranteed to be the global optimal solution [BV04]. However, applying KKT conditions could be impractical as it leads to an exponential computation complexity.

Theorem 4.3. *Solving the single-core mixed-criticality energy minimization problem directly by KKT conditions requires solving $2^{2|\tau_{\text{LO}}|+4|\tau_{\text{HI}}|+4}$ systems of non-linear equations.*

Proof. Notice that for the single-core problem as formulated in Theorem 4.1, only inequality constraints exist. Thus, we need to introduce only the slack variables μ_j in each of the complementary slackness equations $\mu_i g_i(z) = 0$, where at least one of μ_i and $g_i(z)$ must be 0. With m such conditions, there would be 2^m possible cases ($\mu_i = 0$ or $g_i(z) = 0$). For constraints (4.8) - (4.10), we need to introduce four slack variables (in constraint (4.10) we have to consider both upper and lower bounds). According to constraint (4.11), for each HI criticality task, we have to introduce two slack variables for its frequency in each system mode, leading to a total of $4|\tau_{\text{HI}}|$ slack variables. Similarly, we have $2|\tau_{\text{LO}}|$ slack variables for LO criticality tasks (they are not executed in HI mode). This would lead to $2^{2|\tau_{\text{LO}}|+4|\tau_{\text{HI}}|+4}$ binary cases to check for the optimal solution, where in each case we have to solve a system of non-linear equations. □

To reduce the intrinsic high complexity encountered, we proceed towards an in-depth analysis of the energy minimization problem, aiming to reduce the actual search space.

4.3.3 Reduction of Search Space

4.3.3.1 Optimality Condition in f_i^{χ}

We can first reduce the frequency search space by proving that the system energy consumption is minimal when all tasks of the same criticality level share the same frequency in each mode. Let us introduce 3 frequency variables $f_{\text{LO}}^{\text{LO}}$, $f_{\text{HI}}^{\text{LO}}$ and $f_{\text{HI}}^{\text{HI}}$, where $f_{\chi_1}^{\chi_2}$ represents the frequency of all χ_1 criticality tasks in χ_2 system mode. Formally, we have the following result.

Theorem 4.4. *For the single-core mixed-criticality energy minimization problem as specified in Theorem 4.1, in an optimal solution all tasks of the same criticality level share the same frequency in each mode, i.e.,*

$$\forall \tau_i \in \tau_{\text{LO}} : f_i^{\text{LO}} = f_{\text{LO}}^{\text{LO}};\ \ \forall \tau_i \in \tau_{\text{HI}} : f_i^{\text{LO}} = f_{\text{HI}}^{\text{LO}} \wedge f_i^{\text{HI}} = f_{\text{HI}}^{\text{HI}}.$$

Proof. First, let us consider only HI mode energy and show that all HI criticality tasks should share the same execution frequency in HI mode in an optimal solution. Likewise, we can prove similar statements for LO mode task frequencies.

Considering only two HI criticality tasks τ_i and τ_j in the system, we prove using KKT conditions that they share the same execution frequency; by induction, this will hold for any pair of HI criticality tasks and our statement follows.

For our proof here, we denote HI mode utilizations of τ_i and τ_j on base frequency f_b as u_i and u_j ($u_i = \frac{C_i(\text{HI})}{T_i}$, $u_j = \frac{C_j(\text{HI})}{T_j}$), respectively. Using equation (4.5), we can compute HI mode energy as

$$\begin{aligned} E_{\text{HI}} = w_{\text{HI}} \left(u_i \frac{f_b}{f_i^{\text{HI}}} \left(P_s + \beta (f_i^{\text{HI}})^{\alpha} \right) \right) \\ + w_{\text{HI}} \left(u_j \frac{f_b}{f_j^{\text{HI}}} \left(P_s + \beta (f_j^{\text{HI}})^{\alpha} \right) \right). \end{aligned} \tag{4.12}$$

Let u_{i+j} be the allowed total HI criticality system utilization after any DVFS strategy. In equation (4.12), f_i^{HI} and f_j^{HI} are the variables to be determined, such that while applying DVFS to minimize energy E_{HI}, $u_i \frac{f_b}{f_i^{\text{HI}}} + u_j \frac{f_b}{f_j^{\text{HI}}} \leq u_{i+j}$ so that system schedulability is preserved. We find the

optimal f_i^{HI} and f_j^{HI} using KKT conditions. Let us first summarize the above problem as

$$\textbf{minimize} \text{ objective (4.12)}, \quad \textbf{s.t.} \quad u_i \frac{f_b}{f_i^{\text{HI}}} + u_j \frac{f_b}{f_j^{\text{HI}}} \leq u_{i+j}. \tag{4.13}$$

Based on Theorem 4.2, let us combine the inequality constraint with the energy objective and introduce a KKT multiplier (μ). Then, objective (4.12) is at a minimum when

$$\nabla_{(f_i^{\text{HI}}, f_j^{\text{HI}})} E_{\text{HI}} + \mu \nabla_{(f_i^{\text{HI}}, f_j^{\text{HI}})} g\left(f_i^{\text{HI}}, f_j^{\text{HI}}\right) = 0, \tag{4.14}$$

$$\begin{aligned} \textbf{s.t.} \quad & g\left(f_i^{\text{HI}}, f_j^{\text{HI}}\right) = u_i \frac{f_b}{f_i^{\text{HI}}} + u_j \frac{f_b}{f_j^{\text{HI}}} - u_{i+j} \leq 0; \\ & \mu \left(u_i \frac{f_b}{f_i^{\text{HI}}} + u_j \frac{f_b}{f_j^{\text{HI}}} - u_{i+j} \right) = 0 \wedge \mu \geq 0. \end{aligned} \tag{4.15}$$

Under KKT, relations (4.14) and (4.15) must both hold. By solving equality constraint (4.14) further, we have

$$\begin{pmatrix} \frac{\partial(\cdot)}{\partial f_i^{\text{HI}}} \\ \frac{\partial(\cdot)}{\partial f_j^{\text{HI}}} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{4.16}$$

$$\begin{aligned} \Rightarrow w_{\text{HI}} f_b u_i (-P_s (f_i^{\text{HI}})^{-2} + \beta(\alpha - 1)(f_i^{\text{HI}})^{\alpha-2}) &= \mu f_b u_i (f_i^{\text{HI}})^{-2} \\ \Leftrightarrow \beta(\alpha - 1)(f_i^{\text{HI}})^{\alpha} &= P_s + \frac{\mu}{w_{\text{HI}}} \\ \Leftrightarrow f_i^{\text{HI}} &= \left(\frac{P_s + \frac{\mu}{w_{\text{HI}}}}{\beta(\alpha - 1)} \right)^{1/\alpha}. \end{aligned} \tag{4.17}$$

and similarly,

$$\begin{aligned} \Rightarrow w_{\text{HI}} f_b u_j (-P_s (f_j^{\text{HI}})^{-2} + \beta(\alpha - 1)(f_j^{\text{HI}})^{\alpha-2}) &= \mu f_b u_j (f_j^{\text{HI}})^{-2} \\ \Leftrightarrow \beta(\alpha - 1)(f_j^{\text{HI}})^{\alpha} &= P_s + \frac{\mu}{w_{\text{HI}}} \\ \Leftrightarrow f_j^{\text{HI}} &= \left(\frac{P_s + \frac{\mu}{w_{\text{HI}}}}{\beta(\alpha - 1)} \right)^{1/\alpha}. \end{aligned} \tag{4.18}$$

From equations (4.17) and (4.18), we obtain $f_i^{\text{HI}} = f_j^{\text{HI}}$ for which objective (4.12) is minimized. Hence, both tasks share the same execution frequency. By induction, any pair of HI criticality tasks share the same frequency and our statement holds. Similarly, it can be proved that all LO criticality tasks in LO system mode share the same execution frequency and all HI criticality tasks in LO system mode share the same execution frequency. □

In short, Theorem 4.4 is derived by applying KKT conditions separately to several subproblems of our single-core problem. It is also interesting to note that the core of our proof as shown for formulation (4.13) actually deals with a single criticality system. Thus, we have a useful by-product of the above proof: If we have only a single criticality in the system and there is no uncertain workload (i.e., only one level of task WCETs), then all tasks should run on the same frequency to minimize energy under EDF scheduling and the power model in this paper. We shall apply this result to a corner case in Section 4.5.1.

Using Theorem 4.4, the search space in task frequencies can be greatly reduced and the energy objective (4.3) and (4.4) can be simplified as follows:

$$\begin{aligned} E_{\mathrm{LO}} =& w_{\mathrm{LO}} f_b U_{\mathrm{LO}}^{\mathrm{LO}} (P_s / f_{\mathrm{LO}}^{\mathrm{LO}} + \beta (f_{\mathrm{LO}}^{\mathrm{LO}})^{\alpha-1}) \\ &+ w_{\mathrm{LO}} f_b U_{\mathrm{HI}}^{\mathrm{LO}} (P_s / f_{\mathrm{HI}}^{\mathrm{LO}} + \beta (f_{\mathrm{HI}}^{\mathrm{LO}})^{\alpha-1}), \\ E_{\mathrm{HI}} =& w_{\mathrm{HI}} f_b U_{\mathrm{HI}}^{\mathrm{HI}} (P_s / f_{\mathrm{HI}}^{\mathrm{HI}} + \beta (f_{\mathrm{HI}}^{\mathrm{HI}})^{\alpha-1}). \end{aligned} \tag{4.19}$$

In the special case when we only minimize the system LO mode energy, i.e., $w_{\mathrm{LO}} = 1$, we can further show that operating at the maximum frequency in HI mode is optimal to reduce energy. Formally, we have the following result.

Theorem 4.5. *If only system* LO *mode energy* E_{LO} *is considered, then setting* $f_{\mathrm{HI}}^{\mathrm{HI}}$ *to the maximum frequency* $f_{\max}$ *is optimal in reducing system energy.*

Proof. According to Corollary 4.1, if we increase the system operating frequency in HI mode, then the system LO mode energy can be decreased while still satisfying system schedulability. Therefore, we can simply set the system HI mode frequency to the maximum in order to save LO mode energy. □

Theorem 4.5 introduces another interesting property of the "Run and Be Safe" model we proposed in Section 2.4 – if the system LO mode is more likely, then speeding up the HI mode helps in reducing the expected system energy consumption.

4.3.3.2 Optimality Condition in x

We continue to derive the necessary conditions in x (the deadline shortening factor) for a single-core solution to be optimal. We find this by relating the choice of x with the energy consumption in both system modes.

According to Theorem 4.4, let us denote the frequencies in an optimal solution as $f_{\mathrm{LO\,opt}}^{\mathrm{LO}}$, $f_{\mathrm{HI\ opt}}^{\mathrm{LO}}$ and $f_{\mathrm{HI\,opt}}^{\mathrm{HI}}$. Let us further define K, L and M as

follows:

$$K = \sum_{\chi_i=\text{HI}} \frac{C_i(\text{LO}) f_b}{T_i}, \qquad L = \sum_{\chi_i=\text{LO}} \frac{C_i(\text{LO}) f_b}{T_i},$$
$$M = 1 - \sum_{\chi_i=\text{HI}} \frac{(C_i(\text{HI}) - C_i(\text{LO})) f_b}{T_i f^{\text{HI}}_{\text{HI opt}}}. \tag{4.20}$$

Using Corollary 4.1, we can derive the lower and upper bounds of x in an optimal solution as

$$x_{\text{LBopt}} = \frac{\frac{U^{\text{LO}}_{\text{HI}}}{f^{\text{LO}}_{\text{HI opt}}} f_b}{1 - \frac{U^{\text{LO}}_{\text{LO}}}{f^{\text{LO}}_{\text{LO opt}}} f_b} = \frac{K / f^{\text{LO}}_{\text{HI opt}}}{1 - L / f^{\text{LO}}_{\text{LO opt}}}, \tag{4.21}$$

$$x_{\text{UBopt}} = \left[\!\!\left[\frac{1 - \frac{U^{\text{LO}}_{\text{HI}}}{f^{\text{LO}}_{\text{HI opt}}} f_b - \frac{U^{\text{HI}}_{\text{HI}} - U^{\text{LO}}_{\text{HI}}}{f^{\text{HI}}_{\text{HI opt}}} f_b}{\frac{U^{\text{LO}}_{\text{LO}}}{f^{\text{LO}}_{\text{LO opt}}} f_b} \right]\!\!\right]^1$$
$$= \left[\!\!\left[\frac{M - K / f^{\text{LO}}_{\text{HI opt}}}{L / f^{\text{LO}}_{\text{LO opt}}} \right]\!\!\right]^1, \tag{4.22}$$

where $[\![a]\!]^c = \min(a, c)$. Now, we consider choosing the optimal deadline scaling factor x_{opt}. To ensure schedulability, it must follow that, $x_{\text{LBopt}} \leq x_{\text{opt}} \leq x_{\text{UBopt}}$ (see Corollary 4.1). From relations (4.21) and (4.22), we observe that $f^{\text{LO}}_{\text{LO opt}}$ or $f^{\text{LO}}_{\text{HI opt}}$ can be decreased by increasing x_{LBopt} or decreasing x_{UBopt}, thus minimizing LO mode energy[1]. Similarly, $f^{\text{HI}}_{\text{HI opt}}$ can be decreased by decreasing x_{UBopt} to save energy in HI mode. As a result, as long as task frequencies are not minimal ($f_{\min}$) and $x_{\text{LBopt}} \neq x_{\text{UBopt}}$, we can reduce either LO mode frequencies or HI mode frequencies to bring the lower and upper bounds of x closer. This process can only stop if (i) all task frequencies are already lowered to $f_{\min}$ or (ii) $x_{\text{LBopt}} = x_{\text{UBopt}}$. A visualization of this process is given in Figure 4.2.

Note that the above argument still holds even if we fix the system HI mode frequency $f^{\text{HI}}_{\text{HI}}$ to a constant. We summarize our observations on this necessary optimality condition using the following theorem.

[1]Recall that $f_{\min} \geq f_{\text{crit}}$ and reducing frequency is always beneficial in saving total system energy (static and dynamic).

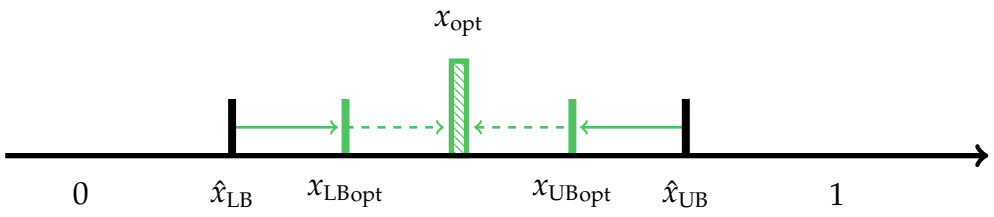


Figure 4.2: Bounds of x in an optimal solution

Theorem 4.6. *An optimal solution to our single-core problem as formulated in Theorem 4.1 exists in one of the two following cases:*

1 *Extreme case: task frequencies satisfy* $f_{\text{LO}}^{\text{LO}} = f_{\text{HI}}^{\text{LO}} = f_{\text{HI}}^{\text{HI}} = f_{\min}$.

2 *Equilibrium case: deadline shortening factor satisfies* $x_{\text{LBopt}} = x_{\text{opt}} = x_{\text{UBopt}}$.

Proof. This directly follows from the discussion above. □

4.3.4 Optimal Solution with KKT

Using Theorem 4.4 and Theorem 4.6, we have a much reduced search space. By adding the optimality condition on task frequencies (Theorem 4.4) to our convex formulation in Theorem 4.1, we can reduce the number of frequency variables from $|\tau_{\text{LO}}|+2|\tau_{\text{HI}}|$ to 3, while preserving the convexity of the problem formulation. Consequently, we can apply KKT conditions to this simplified problem. Using a similar analysis as in Theorem 4.3, we find that 2^{10} systems of non-linear equations need to be solved in order to find the global optimal solution instead of $2^{2|\tau_{\text{LO}}|+4|\tau_{\text{HI}}|+4}$ for our original formulation.

Directly adding the optimality condition on the task deadline scaling factor x, however, will lead to a non-convex problem formulation. According to relations (4.21) and (4.22), the equality constraints $x_{\text{LBopt}} = x_{\text{opt}}$ and $x_{\text{opt}} = x_{\text{UBopt}}$ involve non-affine functions of task frequencies; this violates the convex programming theory [BV04] where only affine functions are allowed in equality constraints. Nevertheless, one could still apply KKT conditions to find local optimal solutions by adding those two equality constraints. In this case, the benefit is that we need to solve 2^{10} systems of non-linear equations with reduced complexities thanks to the additional equality constraints. Since we only have 4 decision variables (x, $f_{\text{LO}}^{\text{LO}}$, $f_{\text{HI}}^{\text{LO}}$, and $f_{\text{HI}}^{\text{HI}}$), it would be possible to compare all local optimal solutions in order to find the global optima.

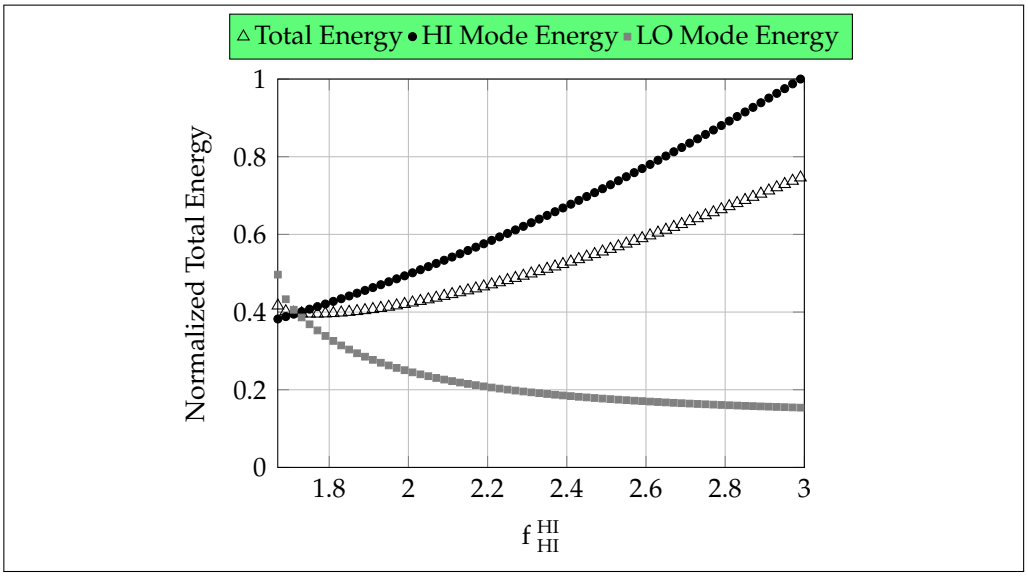


Figure 4.3: Optimal system energy as a function of $f_{\text{HI}}^{\text{HI}}$ for different modes: The following system parameters are used for a better illustration – $f_b = 2.5\text{GHz}$, $[f_{\min}, f_{\max}] = [1, 3]\text{GHz}$, $\alpha = 3$, $\beta = 3\text{W/GHz}^{\alpha}$, $P_s = 4\text{W}$ and $w_{\text{LO}} = 0.3$.

4.4 A Simple & Effective Single-Core Heuristic

As shown in Section 4.3, even by limiting the problem search space without loss of optimality, applying the well-known KKT approach to the single-core energy minimization problem still incurs high complexity, i.e., 2^{10} systems of non-linear equations. We proceed now to develop a computationally simple yet effective heuristic solution to this problem.

4.4.1 Intuition

As already discussed, there exists a trade-off between energy consumptions in different system modes. With increasing $f_{\text{HI}}^{\text{HI}}$, system energy consumption in HI mode will be increased, see equation (4.19). However, this could generate more system slack in LO mode, allowing reduction of LO mode task frequencies to reduce LO mode energy. In other words, increasing $f_{\text{HI}}^{\text{HI}}$ could potentially increase the optimal upper bound of the scaling factor x_{UBopt} as presented in relation (4.22). Due to the equilibrium optimality condition (Theorem 4.6), this could enable us to increase the optimal lower bound of x by decreasing task frequencies in LO mode, see relation (4.21). Similarly, increasing LO mode task frequencies could potentially decrease HI mode system energy.

Moreover, one of our main intuitions is that the second order differentials of both LO and HI mode minimal energy are non-negative with respect to $f_{\text{HI}}^{\text{HI}}$, see equation (4.19). According to equations (4.1)

Algorithm 4.1: A simple and effective single-core heuristic

input : τ, f_b, $f_{\min}$, $f_{\max}$, w_{LO} and w_{HI}
output: $f^{\text{LO}}_{\text{LO opt}}$, $f^{\text{LO}}_{\text{HI opt}}$, $f^{\text{HI}}_{\text{HI opt}}$, x_{opt}

if System feasible when $f^{\text{LO}}_{\text{LO}} = f^{\text{LO}}_{\text{HI}} = f^{\text{HI}}_{\text{HI}} = f_{\max}$ according to Corollary 4.1 **then**
  if System feasible when $f^{\text{LO}}_{\text{LO}} = f^{\text{LO}}_{\text{HI}} = f^{\text{HI}}_{\text{HI}} = f_{\min}$ according to Corollary 4.1 **then**
    $f^{\chi_2}_{\chi_1} \leftarrow f_{\min}\ \forall\chi_1\forall\chi_2$;
  else
    Determine the feasible range of $f^{\text{HI}}_{\text{HI}}$ according to the Corollary 4.1 ($f^{\text{LO}}_{\text{HI}}$ and $f^{\text{LO}}_{\text{LO}}$ set to $f_{\max}$);
    If E' is non-negative at the smallest feasible $f^{\text{HI}}_{\text{HI}}$, set this as $f^{\text{HI}}_{\text{HI opt}}$ (case 1);
    If E' is non-positive at the biggest feasible $f^{\text{HI}}_{\text{HI}}$, set this as $f^{\text{HI}}_{\text{HI opt}}$ (case 2);
    If the above does not hold, we do a binary search to find the $f^{\text{HI}}_{\text{HI}}$ such that $E' = 0$; set this as $f^{\text{HI}}_{\text{HI opt}}$ (case 3);
  end
  return Success;
else
  return Failure;
end

and (4.19), with increasing $f^{\text{HI}}_{\text{HI}}$, dynamic energy consumption in HI mode becomes dominant compared to static energy consumption, leading to a higher rate in energy increase. Furthermore, as $f^{\text{HI}}_{\text{HI}}$ increases, LO mode energy reduces asymptotically to an absolute lower bound when all task frequencies in LO mode are $f_{\min}$. The above observations can be illustrated if we consider the same example from Table 4.2 and plot the optimal energy consumption as a function of $f^{\text{HI}}_{\text{HI}}$ for different modes, as shown in Figure 4.3. We will show later in this section how to derive the optimal energy consumption in each mode individually given $f^{\text{HI}}_{\text{HI}}$. Due to the trade-off between LO and HI mode energy consumptions, there exists a $f^{\text{HI}}_{\text{HI}}$ in the feasible range leading to the minimum overall energy. We aim to find such a $f^{\text{HI}}_{\text{HI}}$ using a simple heuristic.

4.4.2 Heuristic Algorithm

In the extreme case (i.e., all task frequencies are $f_{\min}$), we only need to confirm the system schedulability using Corollary 4.1.

In the equilibrium case, we propose a heuristic approach to find the

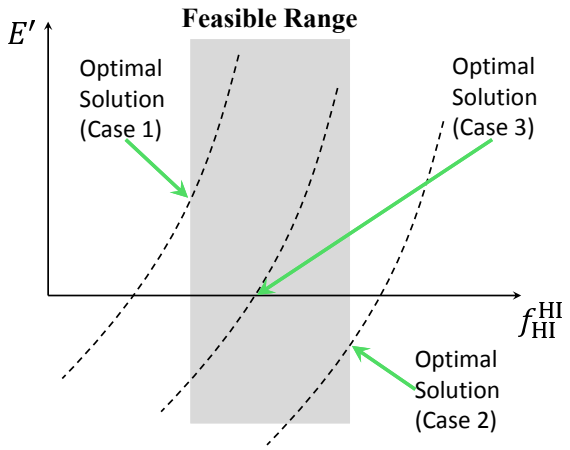


Figure 4.4: Different cases to check in Algorithm 4.1

optimal solution based on the intuitions already discussed in this section. Let us denote with $E_{\chi_{\mathrm{opt}}}$ the minimal χ mode energy consumption with respect to $f_{\mathrm{HI}}^{\mathrm{HI}}$, where $\chi \in \{\mathrm{HI}, \mathrm{LO}\}$. We further denote with E_{opt} the sum of E_{LOopt} and E_{HIopt}, and with E'_{opt} the first order differential of E_{opt}. E'_{opt} would be non-decreasing since the second order differentials of E_{LOopt} and E_{HIopt} are both non-negative. Let the feasible range of $f_{\mathrm{HI}}^{\mathrm{HI}}$ be $[f_{\mathrm{HI\,min}}^{\mathrm{HI}}, f_{\mathrm{HI\,max}}^{\mathrm{HI}}]$, where $f_{\mathrm{HI\,min}}^{\mathrm{HI}}$ is the minimum HI mode frequency required to ensure HI mode schedulability (when LO mode frequencies of tasks are fixed at $f_{\max}$) and $f_{\mathrm{HI\,max}}^{\mathrm{HI}}$ is simply $f_{\max}$. If there exists an optimal $f_{\mathrm{HI}}^{\mathrm{HI}}$ within this range, then it must be in one of these three cases as shown in Figure 4.4.

Case 1 $E'_{\mathrm{opt}} \geq 0$ in the feasible range of $f_{\mathrm{HI}}^{\mathrm{HI}}$: E_{opt} is always increasing, and the optimal solution exists at $f_{\mathrm{HI\,min}}^{\mathrm{HI}}$.

Case 2 $E' \leq 0$ in the feasible range: E_{opt} is always decreasing and the optimal solution exists when $f_{\mathrm{HI\,opt}}^{\mathrm{HI}} = f_{\max}$.

Case 3 The above do not hold: There exists a stationary point $E' = 0$ such that the minimal energy can be achieved. We perform binary search to locate this stationary point.

The above heuristic solution is shown in steps 2-9 in Algorithm 4.1, which only involves logarithmic computation complexity due to binary search. Furthermore, $f_{\mathrm{LO\,opt}}^{\mathrm{LO}}$, $f_{\mathrm{HI\,opt}}^{\mathrm{LO}}$ and x_{opt} are all calculated based on $f_{\mathrm{HI\,opt}}^{\mathrm{HI}}$. We detail this and the calculation of the 1st order differentials in the following.

According to equation (4.19), E_{HI} is a single-variate monotonically increasing function of $f_{\mathrm{HI}}^{\mathrm{HI}}$. Therefore, with fixed $f_{\mathrm{HI}}^{\mathrm{HI}}$, E_{HIopt} equals E_{HI}

and the first order differential can be directly calculated. For E'_{LOopt}, the calculation is less trivial as we still need to find the right LO mode task frequencies leading to the minimal LO mode energy. We achieve this by first deriving E_{LOopt} for any given $f_{\text{HI}}^{\text{HI}}$ based on a similar condition to that in Theorem 4.6; we then calculate E'_{LOopt} empirically using the principal (linear) part [Wik15].

We now explain the derivation of $f_{\text{LO opt}}^{\text{LO}}$, $f_{\text{HI opt}}^{\text{LO}}$, x_{opt} and E_{LOopt} given a fixed value of $f_{\text{HI}}^{\text{HI}}$. We can first prove the LO mode energy is minimal either when all LO mode task frequencies are minimum (f_{min}) or at the equilibrium case, see our comment to Theorem 4.6. For the former case, E_{LOopt} can be directly calculated. For the equilibrium case, we have

$$\frac{K/f_{\text{HI opt}}^{\text{LO}}}{1 - L/f_{\text{HI opt}}^{\text{LO}}} = x_{\text{opt}} = \frac{M - K/f_{\text{HI opt}}^{\text{LO}}}{L/f_{\text{LO opt}}^{\text{LO}}}, \tag{4.23}$$

$$\Leftrightarrow \frac{K/f_{\text{HI opt}}^{\text{LO}} + M - K/f_{\text{HI opt}}^{\text{LO}}}{1 - L/f_{\text{LO opt}}^{\text{LO}} + L/f_{\text{LO opt}}^{\text{LO}}} = x_{\text{opt}},$$

$$\Leftrightarrow x_{\text{opt}} = M,$$

where M is a function of $f_{\text{HI}}^{\text{HI}}$ defined in equation (4.20). We replace here $f_{\text{HI opt}}^{\text{HI}}$ with $f_{\text{HI}}^{\text{HI}}$ as we are fixing the latter. With $x_{\text{opt}} = M$, we can establish a relation between $f_{\text{HI opt}}^{\text{LO}}$ and $f_{\text{LO opt}}^{\text{LO}}$:

$$M = \frac{M - K/f_{\text{HI opt}}^{\text{LO}}}{L/f_{\text{LO opt}}^{\text{LO}}} \Leftrightarrow f_{\text{HI opt}}^{\text{LO}} = \frac{K/M}{1 - L/f_{\text{LO opt}}^{\text{LO}}}. \tag{4.24}$$

Thus, we can represent $f_{\text{HI opt}}^{\text{LO}}$ as a function of $f_{\text{LO opt}}^{\text{LO}}$. Consequently, to find E_{LOopt}, we finally have a single-variate optimization problem in a constrained search space where both $f_{\text{HI opt}}^{\text{LO}}$ and $f_{\text{LO opt}}^{\text{LO}}$ must be drawn from the feasible frequency space. Such a problem can be easily solved by looking at the first order derivatives for continuous optimization. In our implementation, we use standard Mathematica optimization engine [Wol99] to solve this single-variate optimization problem. Once $f_{\text{HI opt}}^{\text{LO}}$ and $f_{\text{LO opt}}^{\text{LO}}$ are derived, we can calculate x_{opt} according to equation (4.23) and E_{LOopt} according to equation (4.19).

In summary, this section provides a heuristic algorithm to solve the single-core energy minimization problem as formulated in Theorem 4.1. Leveraging the optimality conditions in Theorem 4.4 and Theorem 4.6, our heuristic solution is presented in Algorithm 4.1 with further insights discussed in Section 4.4.1.

4.5 Energy Minimization on Multi-Core

In Section 4.3 and Section 4.4, we presented our single-core solutions to the mixed-criticality energy minimization problem. Focusing on partitioned scheduling, we proceed now to extend our single-core techniques to multi-core by further proposing energy-aware task mapping techniques. For all mapping techniques, we assume that unused cores are turned off to conserve energy and task utilizations at the maximum frequency are used when performing mapping. We first briefly discuss two existing mixed-criticality task mapping techniques. We then propose new energy-aware mapping techniques and platform allocation strategies.

4.5.1 Overview of Baruah's and Gu's methods

We first discuss two state-of-the-art mixed-criticality mapping techniques designed to enhance system schedulability rather than energy efficiency.

Baruah's method [Bar14]: In this method, tasks are mapped onto multiple cores using First-Fit (FF) bin-packing, i.e., any task is assigned to an immediate core where it fits first. This is done first for HI criticality tasks and then for LO criticality tasks. In each phase, system utilizations on any core are upper bounded to admit a feasible schedule, i.e., $U_{\text{HI}}^{\text{LO}} + U_{\text{LO}}^{\text{LO}} \leq 3/4 \wedge U_{\text{HI}}^{\text{HI}} \leq 3/4$.

When mapping is successful, local EDF-VD scheduling can be applied independently on each core [BBD+12]. However, due to the fact that Baruah's method only aims at enhancing system schedulability (i.e., using FF to maximize the system utilization on each core), it leaves little room or slack to apply DVFS in order to save energy.

Gu's method [GGDY14]: To further enhance mixed-criticality schedulability on multi-core, this method first assigns all HI criticality tasks onto multi-core by Worst-Fit (WF) bin-packing. The intuition is that HI criticality workload should be balanced across all cores to further admit a fair mix of HI and LO criticality workloads on each core. Mapping of LO criticality tasks is done in the same way as that for Baruah's method. However, Gu's method still allows high utilization of cores (due to greedy mapping of LO criticality tasks), where little room or slack can be explored to save energy by DVFS.

4.5.2 EM3: Energy Minimized Mixed-Criticality Mapping

Both Baruah's and Gu's methods can lead to heavily loaded cores, where little can be done by DVFS to save energy. Thus, we first develop a technique aiming at balancing mixed-criticality workloads on all cores. Achieving this is rather straightforward: We apply WF to map both HI

and LO criticality tasks. Assuming that we have allocated m cores to be used, our proposed method takes the following steps:

1 HI criticality tasks are mapped onto m cores using WF in the order of decreasing utilization, with cumulative HI mode utilization on each core upper bounded by $\frac{3}{4}$.

2 LO criticality tasks are mapped onto m cores using WF in the order of decreasing utilization, with cumulative LO mode utilization on each core upper bounded by $\frac{3}{4}$.

3 DVFS is applied to the mapped task set on each core using the heuristic solution presented in Section 4.4, and all tasks are scheduled using EDF-VD.

4 To find the optimal number of allocated cores, m_{opt}, we repeat steps 1-3 and perform a linear search across all feasible allocations; m_{opt} is the one with the minimum total energy consumption.

Note: (i) If a core contains only LO criticality tasks after task mapping, we ensure that the system utilization does not exceed one under EDF, since we have a single criticality scheduling problem. Nevertheless, we can view this as a special case of the problem studied in this paper. We can prove that all tasks share a common frequency in an optimal solution, see our discussion about Theorem 4.4. Therefore, to minimize energy, we choose a single lowest frequency for a core mapped with only LO criticality tasks, while ensuring that the total system utilization after DVFS is not larger than one.

(ii) If only HI criticality tasks is mapped to a core, then we need to consider an uncertain system workload as those tasks have two levels of WCETs. We view this as a special case of our studied problem where the utilization of LO criticality tasks is zero. We can directly apply the proposed techniques in Section 4.4.

(iii) If possible, we select the minimal set of cores among all allocations leading to the minimal energy consumption. This could minimize the overhead incurred by utilizing additional cores, e.g., non-zero inactive or sleep state energy. We do not explicitly address those overheads in the current chapter.

4.5.3 IM3: Isolated and Energy Minimized Mixed-Criticality Mapping

The above described EM3 method aims to balance mixed-criticality workloads on multi-core to save energy. We will now present a drastically different approach to solve the multi-core energy minimization problem

Algorithm 4.2: Isolated Mixed-criticality Mapping Method

input : τ, π, f_b, $f_{\min}$, $f_{\max}$, w_{LO} and w_{HI}
output: $\mathbb{M}$, $\mathbb{F}$, $\mathbb{X}$

```
Set l̄_0 = ⌈U^LO_LO · f_b/f_max⌉ and h̄_0 = ⌈U^HI_HI · f_b/f_max⌉;
if l̄_0 + h̄_0 ≤ |π| then
    for i in l̄_0 + h̄_0, l̄_0 + h̄_0 + 1, ···, |π| do
        linear search for l̄ and h̄, such that l̄_i ≥ l̄_0 ∧ h̄_i ≥ h̄_0 ∧ l̄_i + h̄_i = i
          and system energy on all utilized cores is minimized;
    end
    Among all i, select l̄_i and h̄_i such that total energy is minimized
      and output M, F, X in this case;
    return Success;
else
    return Failure;
end
```

by isolating tasks of different criticality levels on different cores. This is a common industrial practice to provide independent guarantees to different criticality levels. The expectation is that criticality isolation could also enable energy saving.

Conventionally, when only schedulability is considered, the mainstream research advocates mixing workloads of different criticality levels on each processing core. In this case, smart resource management can be deployed to reconfigure the system under runtime threats by exploring asymmetric guarantees on different criticality levels. As a result, resource efficiency can be enhanced compared to isolation methods [BD16]. However, when energy minimization is the primary goal, mixing workloads of different criticality levels might not be advantageous. On one hand, mixing workloads on all cores can help to achieve global workload balancing, leading to better energy savings. On the other hand, applying mixed-criticality scheduling reduces the maximum attainable system utilization on each core (e.g., $\frac{3}{4}$ for partitioned EDF-VD [Bar14]); this hampers energy saving.

In the new IM3 method, all LO criticality tasks are mapped onto $\bar{l}$ cores and all HI criticality tasks are mapped onto another $\bar{h}$ cores, such that $2 \leq (\bar{l} + \bar{h}) \leq |\pi|$, where $|\pi|$ is the total number of cores available. Both LO and HI criticality tasks are mapped onto their dedicated cores, using WF in the order of decreasing utilization. The choice of $\bar{l}$ and $\bar{h}$ depends on the total utilization of LO and HI criticality tasks, respectively. We apply a simple heuristic to find the best $\bar{l}$ and $\bar{h}$ such that the total normalized energy consumption is minimal, as presented in Algorithm 4.2.

In Algorithm 4.2, we first set the minimum number of required cores for LO criticality tasks to be schedulable as $\lceil U_{\text{LO}}^{\text{LO}} \frac{f_b}{f_{\max}} \rceil$ (similarly $\lceil U_{\text{HI}}^{\text{HI}} \frac{f_b}{f_{\max}} \rceil$ for HI criticality tasks). Next, we perform linear search to find the allocations of LO criticality and HI criticality cores, such that the system energy on all allocated cores is minimal. We select in the end the total core allocation and its corresponding $\bar{l}$ and $\bar{h}$, such that energy is minimal across all possible allocations.

Note that, the total system energy is computed using equation (4.19) for all core allocations and task mappings. Since LO and HI criticality tasks are spatially isolated, we need to deal with the case that each core has tasks of the same criticality level. This has already been discussed in Section 4.5.1. Furthermore, due to isolation, it is worth pointing out that there is no needed to consider weight factors for LO criticality tasks as they are not abandoned. However, this is required for HI criticality tasks as two operation modes exist, and those tasks can have mode-dependent frequencies to minimize the overall energy.

4.6 Evaluation

We evaluate now our proposed energy minimization techniques for mixed-criticality systems with both randomly generated synthetic task sets and a real-life Flight Management System (FMS). Assuming synthetic task sets, we first present the results for single-core and then for multi-core platforms. Next, we evaluate the energy saving of a realistic FMS on a dual-core platform. Last, we evaluate the proposed task mapping techniques in terms of multi-core schedulability for synthetic task sets.

4.6.1 Experiment Setup for Synthetic Task Sets

To validate our proposed techniques in a general setting, we adopt the task generator as shown in Section 2.3.3.1 to generate 100 random feasible task sets at each data point. Our task generator is slightly modified in that LO mode utilizations of LO and HI criticality tasks are drawn from two different ranges; this is performed to better control the utilizations of LO and HI criticality tasks. Recall that we use U_{bound} to denote the maximum system utilization across different modes, P_{HI} to represent the probability that a task is of HI criticality level, and r to represent the ratio of C_i(HI) to C_i(LO) for any HI criticality task. Additionally, the task generator is controlled by the following parameters:

- U_{LO}: the system LO mode utilization.

- $[U_{l-}, U_{l+}]$: utilization of any LO criticality task is uniformly drawn from this range.
- $[U_{h-}, U_{h+}]$: LO mode utilization of any HI criticality task is uniformly drawn from this range.

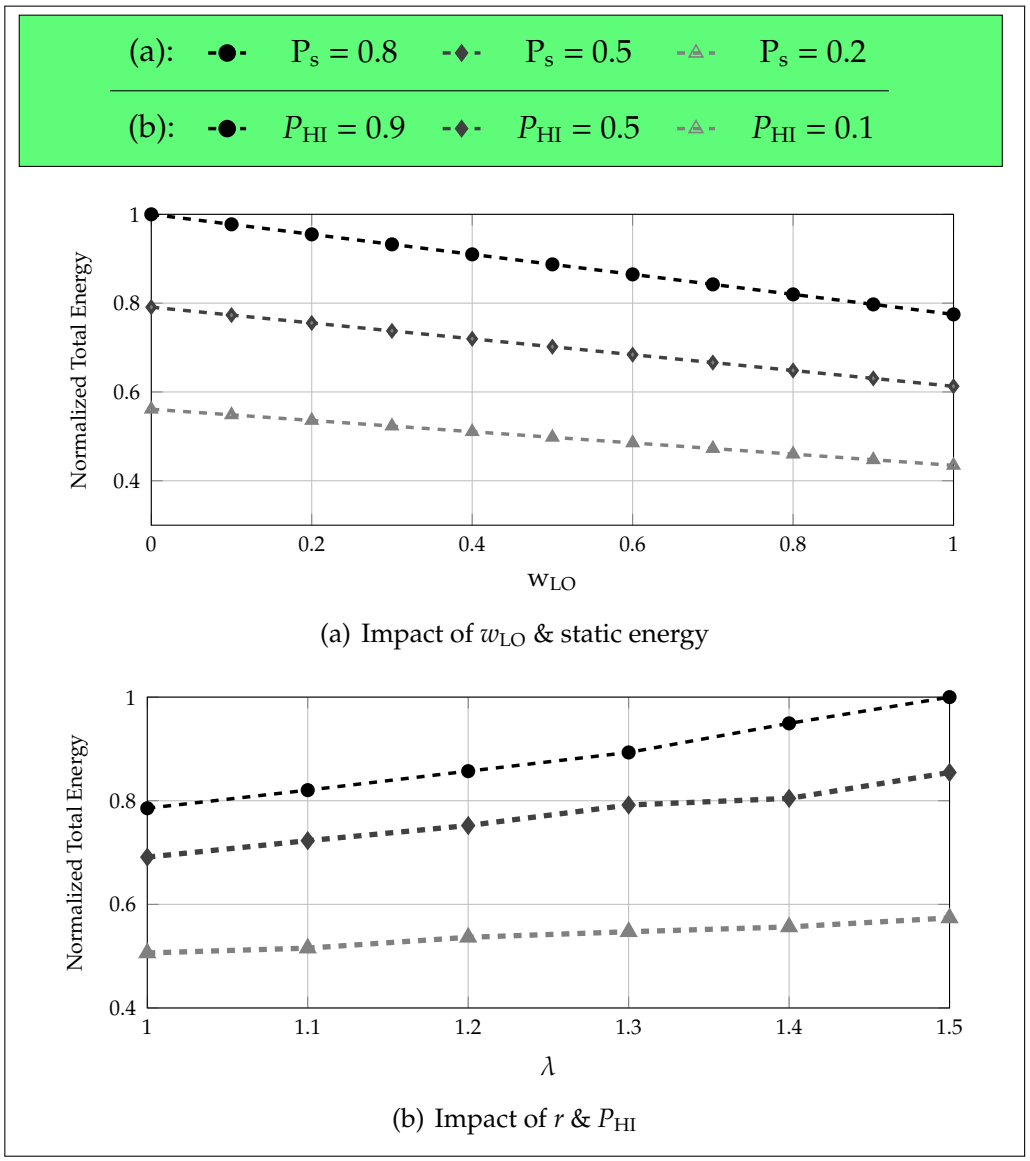
We assume that each core on the platform is DVFS-capable and we apply our proposed techniques to the synthetic task sets to minimize energy; we calculate the energy consumption averaged over all task sets at each data point. All results are normalized such that their maximum in each figure is one. For our experiments on multi-core platforms, we assume that unused cores are turned off to further save energy. Note that, our implemented single-core heuristic as shown in Section 4.4 yields almost identical results when compared to the standard Mathematica optimization engine [mat]. However, the latter is faster as it is highly optimized. For this reason, all optimizations on single-cores are done with Mathematica under our problem formulation (Theorem 4.1) and optimality criteria (Section 4.3).

4.6.2 Single-Core Evaluation

We consider $\{f_{\min}, f_b, f_{\max}\} = \{0.55, 0.85, 1\}$GHz, $P_s = 0.5$W, $\alpha = 2$, $\beta = 1.76$W/GHz$^\alpha$ [NMM+11, PC13b], $U_{\text{LO}} = 0.5$, $[U_{l-}, U_{l+}] = [0.01, 0.05]$, $[U_{h-}, U_{h+}] = [0.05, 0.1]$, $w_{\text{LO}} = w_{\text{HI}} = 0.5$, $P_{\text{HI}} = 0.5$ and $r = 1.5$, unless stated otherwise.

Impact of weight factors: First, we show the impact of weight factors on energy minimization for different system modes. We calculate the normalized total energy with different weight factors w_{LO} (and $w_{\text{HI}} = 1 - w_{\text{LO}}$) in the range $[0, 1]$ in steps of 0.1. Our results are shown in Figure 4.5(a). As we can observe, the choice of weight factors can greatly affect the potential energy saving. In particular, the minimal expected energy decreases by ~20% with w_{LO} increasing from 0 to 1. The reason is that the system utilization in HI mode is higher than that in LO mode (~10% more due to task generation parameters). This further implies that the choice of the right weight factors is crucial to the mixed-criticality energy minimization problems – considering either only HI mode ($w_{\text{LO}} = 0$) or LO mode ($w_{\text{LO}} = 1$) would lead to overly optimistic or pessimistic results. In practice, domain knowledge of the designed application would help choosing such weight factors.

Impact of P_s: We continue to show the impact of static power consumption on energy minimization. Our results are presented in Figure 4.5(a) for w_{LO} in the range $[0, 1]$ and $P_s = 0.2, 0.5, 0.8$W. It is evident from the figure that static power consumption plays an important role in our energy minimization problem – with increasing P_s, the achievable minimal energy consumption is increased significantly. For example,



(a) Impact of w_{LO} & static energy

(b) Impact of r & P_{HI}

Figure 4.5: Normalized total energy on a single-core under the impact of weight factors, static power consumption, r and P_{HI}

when $w_{\text{LO}} = 0.5$, the minimal system energy consumption is increased by 27% when P_s increases from 0.5W to 0.8W.

Impact of r: To show the impact of extra workload, we consider $r = \frac{C(\text{HI})}{C(\text{LO})}$ in the range [1,1.5] in steps of 0.1. When $r = 1$, there is no extra workload as HI criticality tasks would never exceed their LO criticality WCETs. With $r > 1$, there is always a timing safety concern for HI criticality tasks as their HI mode WCETs increase. As we see in Figure 4.5(b), with increasing r, the minimal expected energy consumption increases due to the increase of HI mode system workload.

Impact of P_{HI}: Finally, we demonstrate the impact of P_{HI} on energy

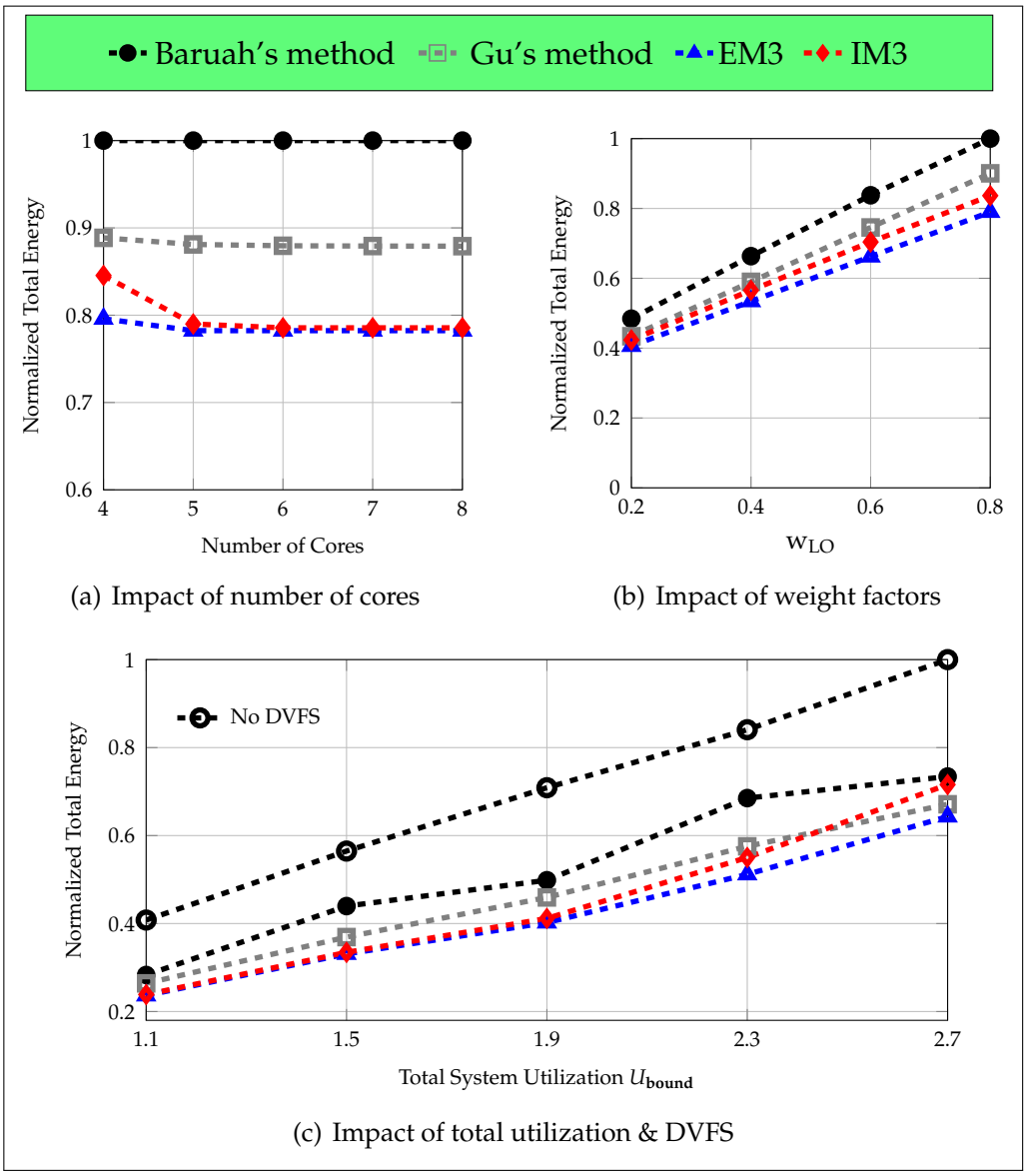


Figure 4.6: Normalized total energy on a multi-core platform under the impact of the number of cores, weight factors, system utilization and P_{HI}

minimization. The results are shown in Figure 4.5(b) for $P_{HI} = 0.1, 0.5, 0.9$. As we can observe, the expected minimal energy increases with increasing P_{HI} – the system LO mode utilization is fixed to 0.5; if P_{HI} increases, more HI criticality tasks are generated, leading to higher extra workload in HI criticality system mode. As a result, the total system workload increases across LO and HI modes, leading to an increase in system energy consumption.

4.6.3 Multi-Core Evaluation

Here, we consider 4 cores, $U_{\text{bound}} = 2.2$, $[U_{l-}, U_{l+}] = [U_{h-}, U_{h+}] = [0.01, 0.1]$, $f_{\min}/f_b/f_{\max} = 0.6/1.2/1.2\text{GHz}$, $\beta = 2\text{W/GHz}^{\alpha}$, $\alpha = 3$, $P_s = 0.8$, $P_{\text{HI}} = 0.2$, $r = 1.5$ and $w_{\text{LO}} = w_{\text{HI}} = 0.5$ in all cases, unless stated otherwise. It should be noted that weight factors are not necessary for LO criticality tasks on a core mapped with only such tasks. However, for a fair comparison with other cases, we include the energy consumptions of LO criticality tasks only in LO mode and use the same equation (4.19) to calculate the system energy on each core. We obtain our results by first simulating different task mapping techniques on the generated task sets. We then apply our single-core method to minimize energy on all cores and to calculate the total normalized energy.

Impact of the number of cores: We obtain our results by performing experiments on 4 to 8 cores and show our results in Figure 4.6(a). From the figure we can see that EM3 and IM3 achieve close energy savings. They save around 20%/10% more energy than Baruah's/Gu's method for 5 cores. The energy consumption for Baruah's method is constant – in all cases, it employs First-Fit method to map tasks, leading to 4 utilized cores and high utilizations on those cores. This decreases the chance to down-scale frequency on active cores to save energy. In Gu's method, only LO criticality tasks are mapped using First-Fit, whereas HI criticality tasks are mapped using Worst-Fit to balance their workload on all cores. This leads to better balanced workload across cores, as well as better energy saving than Baruah's method. Our proposed EM3 method performs load balancing for both criticality levels, leading to the highest energy saving.

Since the IM3 method achieves almost comparable energy savings to that of the EM3 method, which matches our intuition in designing IM3 (see Section 4.5). This leads us to the conclusion that criticality isolation is a good option when minimizing system energy if the number of utilized cores is not a constraint; it provides one additional benefit of providing independence between different criticality levels. Lastly, with increasing number of cores, the minimal system energy for all methods tends to decrease as more cores can be used to balance workload.

Impact of weight factors: We evaluate our methods for different weight factors and show our results in Figure 4.6(b). As we can observe, for $w_{\text{LO}} \in [0.2\,,\, 0.8]$ in steps of 0.2, EM3 always performs best in energy saving and IM3 follows. In addition, as w_{LO} increases, the total energy consumption also increases. This is because on average the LO mode system utilization is higher than the HI mode system utilization for the generated task sets. Hence, when the LO mode weight factor increases, the total energy depends more on LO mode energy and increases.

Impact of system utilization: Clearly, with increasing system

Table 4.3: FMS task set parameters

τ	C_i(LO) (ms)	C_i(HI) (ms)
τ_1	15	21
τ_2	25	36
τ_3	16	22
τ_4	20	28
τ_5	20	35
τ_6	17	24
τ_7	15	21
τ_8	100	100
τ_9	180	180
τ_{10}	140	140
τ_{11}	100	100

utilization, the system operates longer, which increases the achievable minimal energy. We evaluate this by increasing the system utilization for generated task sets from 1.1 to 2.7 in steps of 0.4. The obtained results are shown in Figure 4.6(c). For very low system utilizations – independent of task mapping techniques – all tasks can execute close to f_{min} in both system modes and we observe minor or no differences among the different methods. This trend changes when increasing the system utilization. At higher system utilizations, the EM3 method performs constantly best as it balances both HI and LO mode utilizations across all cores. However, the performance of the IM3 method in energy saving decreases as the system utilization increases – at high system utilizations, strong criticality isolation leads to heavily loaded HI or LO criticality cores and reduces the potential of applying DVFS to save energy. However, even when $U_{\text{bound}} = 2.7$, our proposed IM3/EM3 method still achieves 28%/36% energy saving compared to when DVFS is not applied.

4.6.4 Evaluation with A Flight Management System

Assuming a dual-core platform, we conducted experiments on a real-world Flight Management System (FMS), which is used for aircraft localization, nearest airport selection and trajectory computation. The task set consists of 11 tasks as shown in Table 2.2 and the configuration of task WCETs is listed in Table 4.3. In all experiments, we assume the processor is DVFS-capable with $f_{\text{min}}/f_b/f_{\text{max}} = 0.6/1.2/1.2$GHz, $\beta =$ 2W/GHz$^\alpha$, $P_s = 0.8$W and $\alpha = 3$.

We apply our proposed EM3 and IM3 methods on the FMS application and summarize our results in Figure 4.7. Indeed, we can observe that

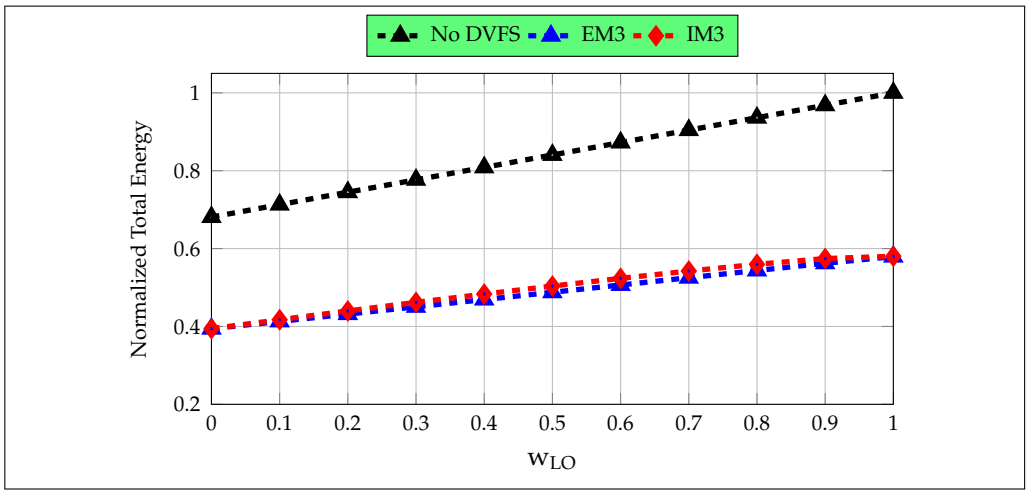


Figure 4.7: Experiment on a flight management system

energy minimization is achieved by applying our technique: around 40% of total energy is reduced by employing DVFS for weight factors in the range $0 - 1$. Additionally, we observe the impact of weight factors on the minimal expected energy: The minimal expected energy increases with increasing w_{LO}, since LO mode utilization is higher than HI mode utilization in the considered task set. Therefore, the expected overall energy increases when more importance is given to minimize LO mode energy. We can also observe that both EM3 and IM3 methods achieve very close energy savings for the FMS application.

4.6.5 Schedulability Evaluation of Mapping Methods

We compare our proposed mapping methods to the state-of-the-art techniques in terms of schedulability, i.e., Baruah's [Bar14] and Gu's [GGDY14] methods. We generate 1000 random task sets at each utilization point similarly to our previous experiments, and map them onto a multi-core platform consisting of 4 cores. We use task generation parameters as follows: $r = 1.25$, $[U_{l-}, U_{l+}] = [0.002, 0.02]$, $[U_{h-}, U_{h+}] = [0.01, 0.1]$ and $P_{\text{HI}} = 0.2$. The number of task sets that are schedulable for different mapping techniques is evaluated with varying system utilization U_{bound} in the range $2.5 - 3$, in steps of 0.1.

We summarize our results in Table 4.4 and Figure 4.8. As we can see, when utilization is low ($U_{\text{bound}} = 2.5$), no matter which mapping technique we use, all task sets are schedulable on 4 cores. As the system utilization increases, IM3 shows the worst performance in terms of schedulability as it cannot mix workload from different criticality levels to improve schedulability. In addition, we observe that Baruah's method is actually slightly better than Gu's method in terms of schedulability. The

Table 4.4: Number of schedulable task sets under different mapping techniques

U_{bound}	Baruah	Gu	EM3	IM3
2.5	1000	1000	1000	1000
2.6	1000	1000	1000	993
2.7	1000	1000	1000	847
2.8	1000	1000	1000	693
2.9	1000	1000	1000	524
3.0	940	925	711	385

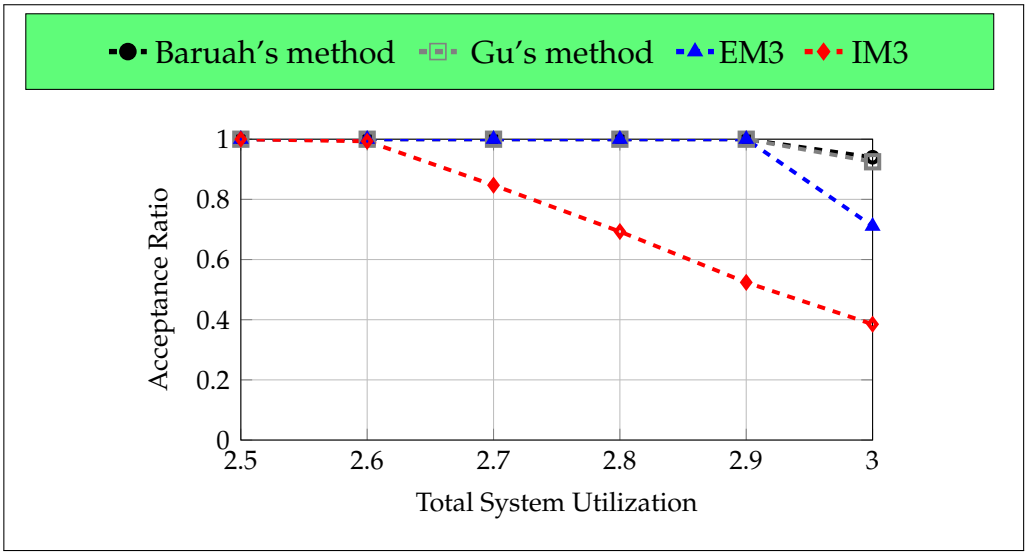


Figure 4.8: Acceptance ratio under partitioned EDF-VD with different mapping methods – system utilizations greater than 3 are not shown as Baruah's, Gu's and EM3 methods stop accepting task sets by construction in this work.

reason is that in Gu's method, further fine tuning on mixing workloads on all cores and on EDF-VD scheduling is performed, which is not considered in this chapter. Last, we find that the EM3 mapping technique has a schedulability performance close to Baruah's and Gu's methods.

4.6.5.1 Summary of Results

For an industry use-case and synthetic task sets, we have demonstrated considerable energy saving on both single-core and multi-core platforms with our proposed energy minimization techniques. In particular, our proposed mapping techniques improve significantly system energy efficiency over state-of-the-art mixed-criticality task mapping techniques. For EM3 this comes with a minor loss of schedulability; for IM3, the gained energy efficiency is at the cost of

a considerable schedulability loss. Our results imply that, if the number of available cores is not constraint in the system, then spatial isolation among different criticality levels is attractive – it is energy-wise efficient and it eases certification thanks to criticality isolation.

4.7 Summary

Mixed-criticality systems are emerging as a significant trend for future automotive and avionic systems. Extending conventional energy minimization techniques to those new systems is of vital importance due to issues related to system cost and thermal safety.

In this chapter, we explored energy minimization for mixed-criticality systems on modern DVFS-capable multi-core platforms. We presented the first solutions known to date for this problem, assuming a general setting where both static and dynamic energy consumption in all system operation modes is considered. To tackle the difficulty in trading off energy consumptions in different modes in order to jointly minimize the overall energy, we first proposed an optimal single-core solution and then a corresponding low computation complexity heuristic. Based on this, we further developed energy-aware task mapping techniques to reduce system energy on multi-core platforms. Experiments were conducted for both a flight management system and synthesized task sets, demonstrating energy savings as high as 36% for multi-core platforms.

Our proposed techniques unified the consideration of mixed-criticality timing guarantee and energy efficiency in tractable solutions, while demonstrating the feasibility of minimizing both static and dynamic energy consumption across different modes to a considerable extent. Furthermore, we have also shown that, strict spatial isolation among different criticality levels – as favored by the industry – achieves almost comparable energy saving to mixing tasks of different criticality levels on each core. This is an interesting and important result: If the number of processors is not a constraint in mixed-criticality systems, spatially isolating tasks of different criticality levels would be energy-wise efficient; it further enables compliance with common safety standards [IEC15, iso11] as strict isolation is often suggested by those standards.

5

Conclusion and Outlook

As a natural consequence of the evolution of computer systems, mixed-criticality systems are emerging in many different application domains [sys]. Those systems commonly host applications of different criticality levels on a same computing platform and show great promises to reduce the overall system cost [BD16]. This thesis is dedicated to the design and optimization of mixed-safety-critical systems, which are commonly found, e.g., in automotive and avionics systems.

A multitude of new challenges can be found in the mixed-criticality era. Most noticeably, "mixing" applications on a common platform with shared resources directly implies interferences among them – an issue that the automotive and avionics industries have tried to avoid in the past few decades [EN16]. To avoid the prohibitively high cost of certifying all applications to the highest criticality level under "mixing", appropriate isolation needs to be guaranteed despite resource sharing. In addition, heterogeneous application requirements as indicated by different criticality levels should be taken into account. To meet those goals, new system models and scheduling techniques are required.

Furthermore, many conventional design issues like fault-tolerance and energy conservation need to be rethought in the mixed-criticality era while considering salient features of these new systems. For example, a common emerging approach for designing mixed-criticality systems is to let them dynamically adapt to runtime threats by asymmetrically protecting critical tasks and sacrificing less critical tasks in such scenarios. While developing methods to achieve reliability and energy saving for mixed-criticality systems, it is important to incorporate such mechanisms and to understand the implication on designing those new systems.

5.1 Contributions

This thesis presents a whole stack of technologies for designing mixed-criticality systems, covering system specification, real-time scheduling, fault-tolerance and energy minimization.

Mixed Criticality Modeling and Scheduling: Chapter 2 introduces first three new mixed-criticality models, all aiming to enhance the service guarantee and flexibility of existing models. The service adaptation model introduces guaranteed degraded service for less critical tasks in urgent scenarios as well as the service resetting time to reset the system to the nominal scenario. The Interference Constraint Graph (ICG) models interferences among tasks of different criticality levels as directed edges among them. The processor over-clocking model takes a hardware perspective and proposes to use processor speedup in order to overcome urgent scenarios. For all proposed models, accompanying scheduling techniques are developed as case studies, while showing considerable improvement over existing methods in terms of system service. Chapter 2 further formalizes the Isolation Scheduling model for mixed-criticality systems, where different criticality levels can only gain mutually exclusive access to the underlying hardware platform. Efficient scheduling techniques are proposed while the schedulability limit of such a model is theoretically investigated.

Fault-Tolerant Mixed-Criticality Design and Analysis: Chapter 3 presents the first mixed-criticality framework, unifying considerations of task deadlines, system reliability and runtime adaptation. Conventional hardening techniques including task re-execution and task replication under transient system faults are applied. The system then dynamically orchestrates resource allocation to different tasks according to their criticality levels and runtime demands. Based on this, safety analysis techniques are developed to bound system reliability, and task deadlines are guaranteed by conventional mixed-criticality scheduling techniques through a problem transformation. Our framework enables the design of safe, schedulable and resource efficient mixed-criticality systems.

Optimizing Energy Efficiency of Mixed-Criticality Systems: Chapter 4 leverages DVFS capability of modern computing platforms to minimize energy consumption of mixed-criticality systems. We reveal fundamental trade-offs involved in this problem and propose optimal as well as heuristic solutions firstly on a single-core. We then present extensions of these results to multi-core with new energy-aware task mapping techniques. Our experimental results on both synthetic and real-life systems demonstrate considerable energy savings. An interesting finding is reported: The industrial practice of spatially isolating criticality levels achieves comparable energy savings to mixing tasks on all cores.

5.2 Possible Future Directions

Broader Application Domains: This thesis has focused on mixed-safety-critical systems emerging in automotive and avionics industries; the notion of criticality is defined from the safety point of view, i.e., a higher criticality indicates a more stringent safety requirement. This definition of criticality would change if other application domains are considered. For example, for Internet systems, it would be interesting to consider mixed security guarantees provided to different information flows. In cloud systems, applications are often characterized with business criticality levels – the criticality of one application indicates its importance to the company's business. Those definitions will naturally affect the guarantee we should make for the corresponding applications and potentially the entire design flow.

Mixed-Criticality Models and Scheduling: To our knowledge, a widely accepted mixed-criticality model for both industry and academia still does not exist. For example, introducing mixed-criticality runtime adaptation is still a controversial topic in the automotive industry [EN16]. Therefore, it is important to extend existing models or propose new ones to close the gap between industry and academia. Such high level models should allow correct system behavior to emerge, and they should allow computationally tractable system design. Meanwhile, they need to be flexible enough to represent many practical systems in order to be widely useful. Last but not least, the designed models should not abstract away important system features that would help the design of mixed-criticality systems, see e.g., Section 2.4.

The scheduling methods we have developed along with the proposed models in Chapter 2 are only proof-of-concepts – they demonstrate that tractable scheduling can be done for those models. Potential further extensions include the consideration of multi-core, memory contention, runtime overhead, etc.

Fault-Tolerance: In Chapter 3, our proposed techniques are built upon the classical mixed-criticality model [BD16], for which possible extensions are discussed in e.g., [GB13]. As future work, it would be interesting and important to extend the methods to incorporate such extensions, e.g., allowing the system to switch back from HI to LO mode while guaranteeing system safety and schedulability. Furthermore, it would be of practical importance to integrate the proposed techniques on sub-systems (e.g., the flight management system in a commercial aircraft) with system wide analysis (e.g., on the level of the aircraft) to achieve certifiable design of industrial mixed-criticality systems.

Energy-Efficiency: In Chapter 4, we have assumed EDF scheduling and focused on the CPU processing energy. It would be interesting

to further consider other scheduling techniques, energy sources (e.g., communication energy) and various system overheads (e.g., timing and energy overheads when switching between different power states). However, we believe that the important findings in Chapter 4 would still hold in such an extension, e.g., trade-off between energy consumptions in different system modes and the benefit of isolated scheduling.

Industrial Case Studies: Perhaps the most effective validation of emerging mixed-criticality techniques is a thorough case study, possibly through collaborations with the industry. The result and certification feedback will provide a meaningful guide for the future development of mixed-criticality systems. This will also provide a chance to test how the segmented mixed-criticality techniques (e.g., in areas of real-time, reliability, energy, etc.) could be unified together and be applicable.

Bibliography

[AB98] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *RTSS*, pages 4–13, 1998.

[ABB09] J. Anderson, S. Baruah, and B. Brandenburg. Multicore operating-system support for mixed criticality. In *Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, 2009.

[AD91] N. Audsley and Y. Dd. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times, 1991.

[ARI03] ARINC. ARINC 653-1 avionics application software standard interface. Technical report, 2003.

[Bar14] Baruah, S. and Chattopadhyay, B. and Li, H. and Shin, I. Mixed-criticality scheduling on multiprocessors. *Real-Time Systems*, 50(1):142–177, 2014.

[BBB09] E. Bini, M. Bertogna, and S. Baruah. Virtual multiprocessor platforms: Specification and use. In *RTSS*, pages 437–446, 2009.

[BBD+11a] S. K. Baruah, V. Bonifaci, G. D'Angelo, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie. Mixed-criticality scheduling of sporadic task systems. In *Algorithms–ESA 2011*, pages 555–566. Springer, 2011.

[BBD11b] S. K. Baruah, A. Burns, and R. I. Davis. Response-time analysis for mixed criticality systems. In *RTSS*, pages 34–43, 2011.

[BBD+12] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *2012 24th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 145–154, July 2012.

[BBDM00] L. Benini, A. Bogliolo, and G. De Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on VLSI*, 8(3):299–316, 2000.

[BBPDM99] L. Benini, A. Bogliolo, A. Paleologo, and G. De Micheli. Policy optimization for dynamic power management. *IEEE Transactions on*

Computer-Aided Design of Integrated Circuits and Systems, 18(6):813–833, Jun 1999.

[BCLS14] S. Baruah, B. Chattopadhyay, H. Li, and I. Shin. Mixed-criticality scheduling on multiprocessors. *Real-Time Systems*, 50(1):142–177, 2014.

[BCPV93] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. In *STOC*, pages 345–354. ACM, 1993.

[BD+13] A. Burns, R. Davis, et al. Mixed criticality on controller area network. In *2013 25th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 125–134. IEEE, 2013.

[BD16] A. Burns and R. Davis. Mixed criticality systems - a review. `www-users.cs.york.ac.uk/burns/review.pdf`, July 2016.

[BDM09] G. Blake, R. G. Dreslinski, and T. Mudge. A survey of multicore processors. *IEEE Signal Processing Magazine*, 26(6):26–37, 2009.

[BDMS+11] V. Bonifaci, G. D'Angelo, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie. Mixed-criticality scheduling of sporadic task systems. *eraerts*, page 204, 2011.

[BF11] S. Baruah and G. Fohler. Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *RTSS*, pages 3–12, 2011.

[BFB15] A. Burns, T. Fleming, and S. Baruah. Cyclic executives, multi-core platforms and mixed criticality applications. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 3–12. IEEE, 2015.

[BHI14] A. Burns, J. Harbin, and L. Indrusiak. A wormhole noc protocol for mixed criticality systems. In *Real-Time Systems Symposium (RTSS), 2014 IEEE*, pages 184–195. IEEE, 2014.

[BLS10] S. Baruah, H. Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *RTAS*, pages 13–22, 2010.

[Boy04] Boyd, S. and Vandenberghe, L. Convex optimization, cambridge university press, isbn: 9780521833783. 2004.

[Bro00] S. Brown. Overview of iec 61508. design of electrical/electronic/programmable electronic safety-related systems. *Computing & Control Engineering Journal*, 11(1):6–12, 2000.

[BV04] S. P. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[BV08] S. Baruah and S. Vestal. Schedulability analysis of sporadic tasks with multiple criticality specifications. In *ECRTS*, pages 147–155, 2008.

[cer] European FP7 CERTAINTY Project. http://certainty-project.eu/.

[CK07] J.-J. Chen and C.-F. Kuo. Energy-efficient scheduling for real-time systems on dynamic voltage scaling (dvs) platforms. In *Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 28–38. IEEE, 2007.

[DB11] R. I. Davis and A. Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Systems*, 47(1):1–40, 2011.

[DFG+11] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu. Memory power management via dynamic voltage/frequency scaling. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 31–40. ACM, 2011.

[DGR04] N. Delgado, A. Q. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30(12):859–872, Dec 2004.

[DO-92] RTCA/DO-178B, Software Considerations in Airborne Systems and Equipment Certification, 1992.

[do11] RTCA/DO-178C, Software Considerations in Airborne Systems and Equipment Certification, 2011.

[DRRG10] F. Dorin, P. Richard, M. Richard, and J. Goossens. Schedulability and sensitivity analysis of multiple criticality tasks with fixed-priorities. *Real-Time Systems*, 46(3):305–331, 2010.

[DVI+02] D. Duarte, N. Vijaykrishnan, M. Irwin, H.-S. Kim, and G. McFarland. Impact of scaling on the effectiveness of dynamic power reduction schemes. In *Proceedings, Computer Design: VLSI in Computers and Processors, IEEE*, pages 382–387, 2002.

[EBA+11] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceedings, Symposium on ICSA*, pages 365–376. IEEE, 2011.

[EBSA+11] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *ISCA*, pages 365–376, 2011.

[EN16] R. Ernst and M. D. Natale. Mixed criticality systems - a history of misconceptions? *IEEE Design Test*, 33(5):65–74, Oct 2016.

[EY12] P. Ekberg and W. Yi. Bounding and shaping the demand of mixed-criticality sporadic tasks. In *ECRTS*, pages 135–144, 2012.

[EY13] P. Ekberg and W. Yi. Bounding and shaping the demand of generalized mixed-criticality sporadic task systems. *Real-Time Systems*, pages 1–39, 2013.

[FLY14] J. Flodin, K. Lampka, and W. Yi. Dynamic budgeting for settling dram contention of co-running hard and soft real-time tasks. In *SIES*, pages 151–159, 2014.

[ga] Genetic Algorithm. `http://ch.mathworks.com/help/gads/genetic-algorithm.html`.

[GAG13] S. Goossens, B. Akesson, and K. Goossens. Conservative open-page policy for mixed time-criticality memory controllers. In *DATE*, pages 525–530, 2013.

[GB10] J. Goossens and V. Berten. Gang ftp scheduling of periodic and parallel rigid real-time tasks. *arXiv preprint arXiv:1006.2617*, 2010.

[GB13] P. Graydon and I. Bate. Safety assurance driven problem formulation for mixed-criticality scheduling. 2013.

[GESY11] N. Guan, P. Ekberg, M. Stigge, and W. Yi. Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems. In *RTSS*, pages 13–23, 2011.

[GGDY14] C. Gu, N. Guan, Q. Deng, and W. Yi. Partitioned mixed-criticality scheduling on multiprocessor platforms. In *DATE*, pages 1–6. IEEE, 2014.

[GLST12] G. Giannopoulou, K. Lampka, N. Stoimenov, and L. Thiele. Timed model checking with abstractions: Towards worst-case response time analysis in resource-sharing manycore systems. In *EMSOFT*, pages 63–72, 2012.

[GSH+15] G. Giannopoulou, N. Stoimenov, P. Huang, L. Thiele, and B. D. de Dinechin. Mixed-criticality scheduling on cluster-based manycores with shared communication and storage resources. *Real-Time Systems*, pages 1–51, 2015.

[GSHT13] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *EMSOFT*, pages 1–15, 2013.

[GSHT14] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele. Mapping mixed-criticality applications on multi-core architectures. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6, March 2014.

[Guo14] C. Guo. Empirical study of energy minimization issues for mixed-criticality systems with reliability constraints. 2014.

[HGA+15] P. Huang, G. Giannopoulou, R. Ahmed, D. B. Bartolini, and L. Thiele. An isolation scheduling model for multicores. In *Proc. of the IEEE Real-Time Systems Symposium (RTSS)*, San Antonio, Texas, USA, Dec 2015.

[HGST13] P. Huang, G. Giannopoulou, N. Stoimenov, and L. Thiele. Service adaptions for mixed-criticality systems. Technical Report 350, ETH Zurich, Laboratory TIK, 2013.

[HGST14] P. Huang, G. Giannopoulou, N. Stoimenov, and L. Thiele. Service adaptions for mixed-criticality systems. In *ASP-DAC*, pages 125–130, 2014.

[HKGT14] P. Huang, P. Kumar, G. Giannopoulou, and L. Thiele. Energy efficient dvfs scheduling for mixed-criticality systems. In *Proceedings of the 14th International Conference on Embedded Software*, EMSOFT '14, pages 11:1–11:10, 2014.

[HRH+12] J. Huang, A. Raabe, K. Huang, C. Buckl, and A. Knoll. A framework for reliability-aware design exploration on mpsoc based systems. *Design Automation for Embedded Systems*, 16(4):189–220, 2012.

[HYT14a] P. Huang, H. Yang, and L. Thiele. On the scheduling of fault-tolerant mixed-criticality systems. Technical report, Computer Engineering and Networks Laboratory, ETH Zurich, 2014.

[HYT14b] P. Huang, H. Yang, and L. Thiele. On the scheduling of fault-tolerant mixed-criticality systems. In *Proceedings of DAC*, DAC '14, pages 131:1–131:6, 2014.

[IEC15] IEC. Iec 61508 standard. `OfficialwebsiteoftheIEC:http://www.iec.ch/about/brochures/pdf/technology/functional_safety.pdf`, August 2015. Accessed: 01-08-2015.

[iso11] ISO 26262, Road Vehicles – Functional Safety, 2011.

[KAB+03] N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *computer*, 36(12):68–75, 2003.

[KAZ11] O. Kelly, H. Aydin, and B. Zhao. On partitioned scheduling of fixed-priority mixed-criticality task sets. In *IEEE TrustCom*, pages 1051–1059, 2011.

[KI09] S. Kato and Y. Ishikawa. Gang edf scheduling of parallel task systems. In *RTSS*, pages 459–468, 2009.

[kin] Energy-efficient solutions: Enabling a new generation of applications. `https://cache.nxp.com/files/shared/doc/NEWENERGEFFICWP.pdf`. Accessed: 2016-01-13.

[KT51] H. Kuhn and A. Tucker. Nonlinear programming. In *Second Berkeley Symposium on Mathematical Statistics and Probability*, pages 481–492. University of California Press, 1951.

[Kum14] P. Kumar. *Hard real-time guarantees in cyber-physical systems*. PhD thesis, Diss., Eidgenössische Technische Hochschule ETH Zürich, Nr. 21755, 2014, 2014.

[KYBS14] J.-E. Kim, M.-K. Yoon, R. Bradford, and L. Sha. Integrated modular avionics (ima) partition scheduling with conflict-free i/o for multicore avionics systems. In *COMPSAC*, pages 321–331, 2014.

[KYK+14] S.-h. Kang, H. Yang, S. Kim, I. Bacivarov, S. Ha, and L. Thiele. Static mapping of mixed-critical applications for fault-tolerant mpsocs. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 31:1–31:6, New York, NY, USA, 2014. ACM.

[L+12] M. Lemke et al. Mixed criticality systems. *Information Society and Media Directorate-General*, 2012.

[LB10] H. Li and S. Baruah. Load-based schedulability analysis of certifiable mixed-criticality systems. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 99–108. ACM, 2010.

[LB12] H. Li and S. Baruah. Global mixed-criticality scheduling on multiprocessors. In *ECRTS*, pages 166–175, 2012.

[LFS+10] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt. Dp-fair: A simple model for understanding optimal multiprocessor scheduling. In *ECRTS*, pages 3–13, 2010.

[LJP13] V. Legout, M. Jan, and L. Pautet. Mixed-criticality multiprocessor real-time systems: Energy consumption vs deadline misses. In *First Workshop on Real-Time Mixed Criticality Systems (ReTiMiCS)*, pages 1–6, 2013.

[LL73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[LPG+14] J. Lee, K.-M. Phan, X. Gu, J. Lee, A. Easwaran, I. Shin, and I. Lee. Mc-fluid: Fluid model-based mixed-criticality scheduling on multiprocessors. In *RTSS*, pages 41–52, 2014.

[Mar10] P. Marwedel. *Embedded system design: Embedded systems foundations of cyber-physical systems*. Springer Science & Business Media, 2010.

[mat] Constrained optimization. `https://www.wolfram.com/technology/guide/ConstrainedNonlinearOptimization/`.

[MEA+10] M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos. Mixed-criticality real-time scheduling for multicore systems. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, pages 1864–1871, 2010.

[MOP10] D. McNulty, L. Olson, and M. Peloquin. A comparison of scheduling algorithms for multiprocessors, 2010.

[NHG+16] S. Narayana, P. Huang, G. Giannopoulou, L. Thiele, and R. V. Prasad. Exploring energy saving for mixed-criticality systems on multi-cores. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12. IEEE, 2016.

[NMM+11] A. Nelson, O. Moreira, A. Molnos, S. Stuijk, B. Nguyen, and K. Goossens. Power minimisation for real-time dataflow applications. In *2011 14th Euromicro Conference on Digital System Design (DSD)*, pages 117–124, Aug 2011.

[NRAW11] A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann. Power management architecture of the 2nd generation intel® core™ microarchitecture, formerly codenamed sandy bridge. Hotchips, 2011.

[PA06] I. Puaut and A. Arnaud. Dynamic instruction cache locking in hard real-time systems. In *Proc. of the 14th Int. Conference on Real-Time and Network Systems*. Citeseer, 2006.

[Pat12] R. M. Pathan. Schedulability analysis of mixed-criticality systems on multiprocessors. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 309–320, July 2012.

[PC13a] S. Pagani and J.-J. Chen. Energy efficiency analysis for the single frequency approximation (sfa) scheme. In *IEEE RTCSA*, pages 82–91, 2013.

[PC13b] S. Pagani and J.-J. Chen. Energy efficient task partitioning based on the single frequency approximation scheme. In *IEEE RTSS*, pages 308–318, 2013.

[PK11] T. Park and S. Kim. Dynamic scheduling algorithm and its schedulability analysis for certifiable dual-criticality systems. In *EMSOFT*, pages 253–262, 2011.

[PQnC+09] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero. Hardware support for wcet analysis of hard real-time multicore systems. In *ISCA*, pages 57–68, 2009.

[RLP+11] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. Pret dram controller: bank privatization for predictability and temporal isolation. In *CODES+ISSS*, pages 99–108, 2011.

[saf16] Safeadapt, 2016. http://www.safeadapt.eu/.

[SCM+14] L. Sha, M. Caccamo, R. Mancuso, J.-E. Kim, M.-K. Yoon, R. Pellizzoni, H. Yun, R. Kegley, D. Perlman, G. Arundale, et al. Single core equivalent virtual machines for hard real-time computing on multicore processors. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, November 2014.

[SEL08] I. Shin, A. Easwaran, and I. Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *ECRTS*, pages 181–190, 2008.

[SGTG12] F. Santy, L. George, P. Thierry, and J. Goossens. Relaxing mixed-criticality scheduling strictness for task sets scheduled with fp. In *ECRTS*, pages 155–165, 2012.

[Sud01] K. Sudhir. Competitive pricing behavior in the auto market: A structural analysis. *Marketing Science*, 20(1):42–60, 2001.

[sys] Mixed criticality systems. http://cordis.europa.eu/fp7/ict/embedded-systems-engineering/documents/sra-mixed-criticality-systems.pdf.

[SZ13] H. Su and D. Zhu. An elastic mixed-criticality task model and its scheduling algorithm. In *DATE*, pages 147–152, 2013.

[TAED13] S. Tobuschat, P. Axer, R. Ernst, and J. Diemer. Idamc: A noc for mixed criticality systems. In *RTCSA*, pages 149–156, 2013.

[TB94] K. Tindell and A. Burns. Fixed priority scheduling of hard real-time multi-media disk traffic. *The Computer Journal*, 37(8):691–697, 1994.

[TBW95] K. Tindell, A. Burns, and A. J. Wellings. Calculating controller area network (can) message response times. *Control Engineering Practice*, 3(8):1163–1169, 1995.

[TCN00] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *The 2000 IEEE International Symposium on Circuits and Systems*, volume 4, pages 101–104. IEEE, 2000.

[TSP11] D. Tamas-Selicean and P. Pop. Design optimization of mixed-criticality real-time applications on cost-constrained partitioned architectures. In *RTSS*, pages 24–33, 2011.

[TSP15] D. Tămaş-Selicean and P. Pop. Design optimization of mixed-criticality real-time embedded systems. *ACM Trans. on Embeddded Computing Systems*, 14(3):50:1–50:29, 2015.

[Ves07] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *RTSS*, pages 239–243, 2007.

[VHL14] M. Volp, M. Hahnel, and A. Lackorzynski. Has energy surpassed timeliness? scheduling energy-constrained mixed-criticality systems. In *IEEE RTAS*, pages 275–284, 2014.

[Vog10] T. Vogelsang. Understanding the energy consumption of dynamic random access memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 363–374. IEEE Computer Society, 2010.

[Wik15] Wikipedia. Differential of a function — wikipedia, the free encyclopedia, 2015. [Online; accessed 15-November-2015].

[Wik16a] Wikipedia. Arinc 653 — wikipedia, the free encyclopedia, 2016. [Online; accessed 20-September-2016].

[Wik16b] Wikipedia. Automotive electronics — wikipedia, the free encyclopedia, 2016. [Online; accessed 7-July-2016].

[WKP13] Z. P. Wu, Y. Krish, and R. Pellizzoni. Worst case analysis of dram latency in multi-requestor systems. In *RTSS*, pages 372–383, 2013.

[Wol99] S. Wolfram. *The Mathematica*. Cambridge university press Cambridge, 1999.

[Won08] K. W. Wong. System and method for providing a thermal shutdown circuit with temperature warning flags, November 18 2008. US Patent 7,454,640.

[YMWP14] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *RTAS*, pages 155–166, 2014.

[YWA+11] H. Yun, P.-L. Wu, A. Arya, C. Kim, T. Abdelzaher, and L. Sha. System-wide energy optimization for multiple dvs components and real-time tasks. *Real-Time Systems*, 47(5):489–515, 2011.

[YYP+12] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *ECRTS*, pages 299–308, 2012.

[ZFMS03] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam. An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):236–247, 2003.

[ZHT16] L. Zeng, P. Huang, and L. Thiele. Towards the design of fault-tolerant mixed-criticality systems on multicores. In *the International Conference on Compilers, Architectures and Synthesis For Embedded Systems*, Pittsburgh, PA, USA, Oct 2016.

[ZPKW13] M. Zeller, C. Prehofer, D. Krefft, and G. Weiss. Towards runtime adaptation in autosar. *ACM SIGBED Review*, 10(4):17–20, 2013.

List of Publications

The following list includes publications that form the basis of this thesis. The corresponding chapters are indicated in parentheses.

⋆ corresponding author

Pengcheng Huang, Pratyush Kumar, Nikolay Stoimenov, and Lothar Thiele. **Interference Constraint Graph – A New Specification for Mixed-Criticality Systems.** In *Proceedings of the Emerging Technologies Factory Automation (ETFA).* Cagliari, Italy, September 2013. (Chapter 2)

Pengcheng Huang, Georgia Giannopoulou, Nikolay Stoimenov, and Lothar Thiele. **Service adaptions for mixed-criticality systems.** In *Proceedings of the In Design Automation Conference (ASP-DAC).* Singapore, January 2014. (Chapter 2)

Pengcheng Huang, Pratyush Kumar, Georgia Giannopoulou, Lothar Thiele. **Run and Be Safe: Mixed-Criticality Scheduling with Temporary Processor Speedup.** In *Proceedings of the 2015 Design, Automation and Test in Europe Conference (DATE).* Grenoble, France, March 2015. (Chapter 2)

Pengcheng Huang, Georgia Giannopoulou, Rehan Ahmed, Davide Basilio Bartolini, and Lothar Thiele. **An isolation scheduling model for multicores.** In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS).* San Antonio, USA, December 2015. (Chapter 2)

Pengcheng Huang, Hoeseok Yang, and Lothar Thiele. **On the scheduling of fault-tolerant mixed-criticality systems.** In *Proceedings of the 51st Annual Design Automation Conference (DAC).* San Francisco, USA, June 2014. (Chapter 3)

Luyuan Zeng, Pengcheng Huang⋆, and Lothar Thiele. **Towards the Design of Fault-Tolerant Mixed-Criticality Systems on Multicores.** In *Proceedings of the International Conference on Compilers, Architectures and Synthesis For Embedded Systems (CASES).* Pittsburgh, USA, October 2016. (Chapter 3)

Pengcheng Huang, Pratyush Kumar, Georgia Giannopoulou, Lothar Thiele. **Energy Efficient DVFS Scheduling for Mixed-Criticality Systems.** In *Proceedings of the 14th International Conference on Embedded Software (EMSOFT).* New Dehli, India, October 2014. (Chapter 4)

Sujay Narayana, Pengcheng Huang⋆, Georgia Giannopoulou, Lothar Thiele and R.Venkatesha Prasad. **Exploring Energy Saving for Mixed-Criticality Systems on Multi-cores.** In *Proceedings of 22nd IEEE Real-Time Embedded Technology & Applications Symposium (RTAS).* Vienna, Austria, April 2016. (Chapter 4)

The following list includes publications that are not part of this thesis.

Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, Lothar Thiele. **Scheduling of Mixed-Criticality Applications on Resource-Sharing Multicore Systems.** In *Proceedings of the 11th ACM International Conference on Embedded Software (EMSOFT).* Montreal, Canada, October 2013.

Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, Lothar Thiele. **Mapping Mixed-Criticality Applications on Multi-Core Architectures.** In *Proceedings of the Design, Automation and Test in Europe Conference (DATE).* Dresden, Germany, March 2014.

Devendra Rai, Pengcheng Huang, Nikolay Stoimenov, Lothar Thiele. **An Efficient Real Time Fault Detection and Tolerance Framework Validated on the Intel SCC Processor.** In *Proceedings of the 51st Design Automation Conference (DAC).* San Francisco, US, June 2014.

Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, Lothar Thiele, Benoit Dupont de Dinechin. **Mixed-Criticality Scheduling on Cluster-Based Manycores with Shared Communication and Storage Resources.** In *Real-Time Systems Journal.* May 2015.

Lukas Sigrist, Georgia Giannopoulou, Pengcheng Huang, Andres Gomez, Lothar Thiele. **Mixed-Criticality Runtime Mechanisms and Evaluation on Multicores.** In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS).* Seattle, US, April 2015.

Biao Hu, Kai Huang, Pengcheng Huang, Lothar Thiele and Alois Knoll. **On-the-Fly Fast Overrun Budgeting for Mixed-Criticality Systems.** In *Proceedings of the International Conference on Embedded Software (EMSOFT).* Pittsburgh, US, October 2016.

Romain Jacob, Marco Zimmerling, Pengcheng Huang, Jan Beutel and Lothar Thiele. **Towards Real-time WirelessCyber-physical Systems.** In *Proceedings of the 28th Euromicro Conference on Real-Time Systems (ECRTS).* Toulouse, France, July 2016.

Romain Jacob, Marco Zimmerling, Pengcheng Huang, Jan Beutel and Lothar Thiele. **End-to-end Real-time Guarantees in Wireless Cyber-physical Systems.** In *Proceedings of the 37th IEEE Real-Time Systems Symposium (RTSS).* Porto, Portugal, December 2016.

Stefan Draskovic, Pengcheng Huang and Lothar Thiele. **On the Safety of Mixed-Criticality Scheduling.** In *4th Workshop on Mixed-Criticality.* Porto, Portugal, December 2016.

Georgia Giannopoulou, Pengcheng Huang, Rehan Ahmed, Davide Basilio Bartolini and Lothar Thiele. **Isolation Scheduling on Multicores: Model and Scheduling Approaches.** To appear.

Biao Hu, Lothar Thiele, Pengcheng Huang, Kai Huang, Christoph Griesbeck and Alois Knoll. **FFOB: Efficient Online Mode-Switch Procrastination in Mixed-Criticality Systems, under review.** To appear.

Curriculum Vitæ

Personal Data

Name	Pengcheng Huang
Date of Birth	May 26, 1987 (Chinese lunar calendar)
Citizenship	Chinese

Education

12/2011 – 12/2016	ETH Zurich, Switzerland Computer Engineering and Networks Laboratory Ph.D. under the supervision of Prof. Dr. Lothar Thiele
09/2009 – 10/2011	Technology University of Delft, the Netherlands Master in Computer Engineering
09/2005– 07/2009	Harbin Institute of Technology, China Bachelor in Instrumentation

Professional Experience

12/2011 – 12/2016	ETH Zurich, Switzerland Computer Engineering and Networks Laboratory Graduate research and teaching assistant
10/2010 – 07/2011	ST-Ericsson, Eindhoven, the Netherlands Multicore DSP and Streaming Applications Intern

Honors and Awards

- Top Talent Scholarship, TU Delft, 2009-2011