UNIVERSITY OF HERTFORDSHIRE

Adaptive Mid-term and Short-term Scheduling of Mixed-criticality Systems

Saverio Iacovelli

A thesis submitted to the University of Hertfordshire in partial fulfillment of the requirements of the degree of

Doctor of Philosophy

in Computer Science

September 2018

Abstract

A mixed-criticality real-time system is a real-time system having multiple tasks classified according to their criticality. Research on mixed-criticality systems started to provide an effective and cost efficient a priori verification process for safety critical systems. The higher the criticality of a task within a system and the more the system should guarantee the required level of service for it. However, such model poses new challenges with respect to scheduling and fault tolerance within real-time systems. Currently, mixed-criticality scheduling protocols severely degrade lower criticality tasks in case of resource shortage to provide the required level of service for the most critical ones. The actual research challenge in this field is to devise robust scheduling protocols to minimise the impact on less critical tasks.

This dissertation introduces two approaches, one short-term and the other medium-term, to appropriately allocate computing resources to tasks within mixed-criticality systems both on uniprocessor and multiprocessor systems.

The short-term strategy consists of a protocol named *Lazy Bailout Protocol* (LBP) to schedule mixed-criticality task sets on single core architectures. Scheduling decisions are made about tasks that are active in the ready queue and that have to be dispatched to the CPU. LBP minimises the service degradation for lower criticality tasks by providing to them a background execution during the system idle time. After, I refined LBP with variants that aim to further increase the service level provided for lower criticality tasks. However, this is achieved at an increased cost of either system offline analysis or complexity at runtime.

The second approach, named *Adaptive Tolerance-based Mixed-criticality Protocol* (ATMP), decides at runtime which task has to be allocated to the active cores according to the available resources. ATMP permits to optimise the overall system utility by tuning the system workload in case of shortage of computing capacity at runtime. Unlike the majority of current mixed-criticality approaches, ATMP allows to smoothly degrade also higher criticality tasks to keep allocated lower criticality ones.

Acknowledgements

First of all, I would like to thank the University of Hertfordshire for providing me the funding and infrastructure for my PhD and for giving me the opportunity to do research in Computer Science. Doing research in computing is now a dream that I have realised.

I would like to thank my two supervisors. Firstly, my gratitude goes to my principal supervisor, Dr. Raimund Kirner, for his constant feedback, encouragements and for all what I have learnt so far. Then, I would like also to thank my second supervisor, Dr. Olga Tveretina, for her timely feedbacks and suggestions.

I also thank Dr. Catherine Menon for her useful contributions to my work with her support about safety in mixed-criticality systems.

I also want to thank and remember all my previous colleagues in my office, for lunch and fun in the discussion room, conversations with them about silly as well as technical stuff. An intense part of my life is with them. I thank also all other researchers and students that I have known so far. They all have contributed to enrich the knowledge of my research field, of related areas and of the world in general.

Lastly, but just for order, I thank all my family, irreplaceable reference point of my life. No achievement would had been possible in my life without it.

Contents

List of Figures

List of Tables

Acronyms

AMC	Adaptive Mixed-Criticality
AMC-rtb	Adaptive Mixed-Criticality - Response Time Bound
ATMP	Adaptive Tolerance-based Mixed-criticality Protocol
BP	Bailout Protocol
BPG	Bailout Protocol with Gain time
BPS	Bailout Protocol - Slack time
BPSG	Bailout Protocol - Slack and Gain time
CAPA	Criticality As Priority Assignment
DM	Deadline Monotonic
E-MC	Elastic Mixed-Criticality
EDF-VD	Earliest Deadline First with Virtual Deadline
ER-EDF	Early Release Earliest Deadline First
LBP	Lazy Bailout Protocol
LBPG	Lazy Bailout Protocol with Gain time
LBPS	Lazy Bailout Protocol - Slack time
LBPSG	Lazy Bailout Protocol - Slack and Gain time
RM	Rate Monotonic
RTA	Response Time Analysis
SIL	Safety Integrity Level
SLBP	Soft Lazy Bailout Protocol
SLBPG	Soft Lazy Bailout Protocol with Gain time
SLBPS	Soft Lazy Bailout Protocol - Slack time
SLBPSG	Soft Lazy Bailout Protocol - Slack and Gain time
TRTCM	Tolerance-based Real-Time Computing Model

TUF	Time/Utility Function
UA	Utility Accrual
WCET	Worst-Case Execution Time

Chapter 1

Introduction

This chapter introduces and motivates my research work. It contains introductory descriptions about the research field, my contributions and about how the assessment has been made.

Section 1.1 introduces the research field and background. Section 1.2 describes the new challenges arising with the mixed-criticality scheduling both on uniprocessor and multiprocessor platforms. Section 1.3 contains my research question together with the motivations that justify my research work. Section 1.4 lists and summarises the contributions contained in this dissertation. Section 1.5 describes how the improvement of each contribution has been measured with respect to the state-of-the-art. Section 1.6 lists the papers already published and in writing stage. Finally, Section 1.7 and Section 1.8 contain respectively the structure of the dissertation and the summary of this chapter.

1.1 Mixed-Criticality Real-Time Systems

Real-time computer systems are systems in which the correctness of the system behaviour depends not only on the logical results of computations but also on the physical time at which such results must be provided [2]. As all other computing systems, real-time systems typically provide different services according to their specifications. Such services could consist of one or more tasks that might generate an unbounded sequence of instances, each has to be processed and completed within a specific temporal deadline. Different tasks can be ranked according to different criticality. A mixed-criticality system is a system in which multiple functionalities of different criticalities are implemented and integrated on the same platform. In such systems the real-time properties indicated by the tasks' urgency can lead more critical tasks to not complete within their temporal

deadlines in case of resource shortages.

The task's criticality can be derived from different aspects. One possibility is to express the relative importance or relative utility of different services in a system as their criticality [6]. The term criticality is also very commonly used in the context of functional safety, e.g., to express the relative level of assurance of a system functionality dictated by different development standards for safety critical systems, like DO-178C [7] in the avionics domain, ISO26262 [8] in the automotive domain or IEC 61508 [9] in the automation, as different levels of criticality. Another interpretation of criticality is to indicate the importance of a task related to the strictness of its deadline [10]. The meaning of criticality is still sometimes subject of discussion, however this dissertation does not mandate to any specific procedure for defining criticality levels, as this is an orthogonal issue to the mixed-criticality scheduling itself.

Specific research on mixed-criticality scheduling started in 2007 by Vestal et al. to provide a safe and cost effective a priori verification process for systems with increasingly integration of services on the same platform [11]. The key idea underlying is to construct multiple models of the same platform, each of which true to a different level of confidence. Then, tasks with different criticality are verified at different levels of assurance [12]. Such modular approach allows to avoid the usage of excessively conservative models for verifying less critical tasks, thus reducing the resource under-utilization and the overall verification cost. The system model proposed by Vestal consists of a set of tasks, each may have a set of alternative *Worst-Case Execution Time* (WCET) estimates assured to a different level of confidence. The final aim was that to guarantee the correct completion of highly critical tasks.

1.2 Mixed-Criticality Scheduling

The mixed-criticality scheduling aims to bring the fault handling into real-time scheduling by ensuring a correct completion of higher criticality tasks in case some parts of the system do not conform to the behaviour assumed in the a priori verification. In fact, mixed-criticality scheduling approaches are built on the key concept that tasks should be prioritised not only according to their timing requirements or urgency but also according to the impact that they could have on the overall system in specific situations based on some predefined criterion like importance or safety, usually referred to as criticality. Typically, such situations occur when the resource shortages make not possible to schedule all the task

instances according to the predefined scheduling policy and it becomes necessary to choose which instance to keep running and which to abandon by actually changing their priorities.

As an example, system engineers usually design systems to have task instances that complete within their WCET estimates and that are processed according to real-time scheduling protocols that give priority to task instances with stricter timing requirements, that are mainly represented by arrival rates or deadlines. This approach works well as long as it can be assured that enough resources are available to schedule all tasks. However, in cases where availability of enough resources cannot be guaranteed, traditional real-time scheduling methods miss the flexibility to prioritise the resources to certain tasks. In fact, if some processing elements or memory resource suddenly become unavailable or if some communication line is temporary busy or occupied, some instances could exceed their time threshold. In case highly critical task instances do not complete their execution within it, this situation can lead to a phenomenon called *criticality inversion* in which new incoming low-criticality instances have higher scheduling priority than currently running high-criticality instances [13].

Mixed-criticality scheduling protocols on uniprocessor platforms are designed to manage such resource shortage situations, also referred to as transient faults [14], and hence they are inherently fault tolerant. The active research challenge is that to find ways to effectively combine the resource prioritisation based on criticalities with the scheduling priorities based on real-time constraints in order to minimise the impact on less critical tasks. The basic functioning of mixed-criticality scheduling so far is that as long as enough resources are available, the scheduling priorities are defined by a real-time scheduling protocol. In case of a resource shortage, e.g., a critical instance overrunning its WCET [15], the tasks' criticalities are used as primary criterion to allocate resources.

Most of work on mixed-criticality scheduling on uniprocessor platforms deals with dual-criticality systems, i.e., systems in which there are two levels of criticality, frequently labelled as LO (low criticality) and HI (high criticality). This interest in dual-criticality systems is based on the fact that they represent a reasonable simplification of more complex systems that permit to devise scheduling methods suitable also for systems with multiple criticality by just using only two levels of WCET estimates. The usual approach is to assume for LO tasks only the knowledge of easy to derive optimistic WCET estimates while for HI tasks also a higher level of assurance based on safe upper WCET bounds is assumed.

The research has produced many mixed-criticality scheduling protocols but they mostly degrade very severely the service provided to LO tasks. Such degra-

dation results from dropping a significant amount of their corresponding instances. A form of advanced and adaptive fixed-priority protocol to schedule set of tasks having two criticality levels is the *Bailout Protocol (BP)* [16, 17]. As all previous mixed-criticality protocols, the BP acts as a standard fixed-priority scheduler since enough resources are available and then switches to a restricted execution mode in which the HI tasks are prioritised and the LO task instances are abandoned. The contribution of BP is that to provide an effective and fast recovery mechanism to switch back to the starting execution mode in which all instances can be scheduled. However, the main weakness of BP is that it completely aborts the service provided to LO tasks during resource shortages and such behaviour is not tolerable to make mixed-criticality protocols suitable for industry and acceptable for system engineers.

Furthermore, the increasingly adoption of multi-core computing architectures is leading the mixed-criticality research community to devise systems in which tasks having different criticality can be allocated to different processing elements. Researchers have been working to appropriately manage such systems by devising both ad hoc multi-core hardware architectures [18, 19] and novel scheduling protocols [20, 21]. Multiprocessor platforms raise new issues with regard to mixed-criticality scheduling since the system should minimise the impact of failures of one or more cores at run-time by guraanteeing adequate service first to tasks of higher criticality. A way to handle the sudden unavailability of computing resources is to employ the spatial redundancy. However, the hardware replication should consider the *Size, Weight and Power* (SWaP) constraints of the embedded systems [14]. Another solution to manage such type of faults is via a criticality-aware reallocation of tasks in which the available computing resources have to be assigned first to higher criticality tasks.

1.3 Research Question

Research about optimal allocation of computational resources to mixed-criticality tasks has produced different approaches that sometime are not integrated among them to provide a broader architecture.

A concrete model for mixed-criticality optimisation is the *Tolerance-based Real-Time Computing Model* (TRTCM) [1, 22]. The key idea underlying the TRTCM model is that the utility of services provided by a real-time system can be beneficial even at a degraded level rather than being aborted as long as the smooth degradation occurs within an acceptable minimum threshold. With

regard to this, TRTCM introduces the notion of operational ranges for real-time service requirements of reduced but still acceptable service utility. In case of unexpected resource shortage, such tolerance intervals give to the system a means to decide at runtime what service to degrade or even abort. The TRTCM allows for runtime reconfiguration based on multiple performance parameters and for utility optimisation on both uniprocessor and multiprocessor platforms.

However, the current mixed-criticality scheduling approaches on single core architectures focus on guaranteeing the adequate service level for high criticality tasks in all operating conditions, including the worst-case scenarios, always by degrading lower criticality tasks. With regard to this, an effective fixed-priority mixed-criticality scheduling protocol on uniprocessor platforms is represented by the *Bailout Protocol* (BP) [16, 17]

The aim of my research has been to optimise the utility of mixed-criticality real-time systems in case of resource shortages. TRTCM as well as existing mixed-criticality scheduling protocols represent two different means to achieve such objective. TRTCM achieves this by finding a useful compromise between the high and low critical services according to the goal function and to the system and resource constraints while BP and the related protocols always abandon lower criticality instances that have less impact on the overall system in case of transient faults. However, the TRTCM does not specify any concrete mixed-criticality scheduling protocol while the scheduling methods devised so far are basically short-term strategies that do not consider the usage of mid-term planning. In fact, the TRTCM consists of a model together with a related optimisation problem that specifies, at a high level, what parameters and utilities should be set but currently there is no algorithm to connect the optimisation model to the underlying scheduling protocol. On the other hand, currently no existing mixed-criticality scheduling algorithm is integrated into a higher level model for utility-based mid-term planning.

The research question that has motivated and guided my research is the following:

Is it possible to improve the state of the art of fixed-priority mixed-criticality scheduling and then combine such protocols with the TRTCM model to optimise the utility of mixed-criticality real-time systems in case of resource shortages?

This question can be formulated more precisely by the following three sub-questions:

1. **Is it possible to increase the robustness of existing fixed-priority**

mixed-criticality scheduling protocols?

Currently, real-time mixed-criticality task models are based on the usage of different estimates of one ore more task parameters, e.g., inter-arrival time or worst-case execution time. The higher is the criticality and the more conservative is the task parameter estimate. Most of such scheduling protocols consider set of tasks having two criticality levels, indicated by HI and LO with HI being more critical than LO.

Under such assumptions, the existing fixed-priority mixed-criticality scheduling protocols always assure the correct completion of HI tasks in all operating conditions at the expense of the remaining ones. This can lead to a large amount of LO tasks being abandoned or aborted.

As a result, to increase the robustness of current mixed-criticality methods it is necessary to devise algorithms that allow to successfully schedule more LO tasks.

2. **Is it possible to connect the TRTCM optimisation problem with the mixed-criticality scheduling protocols?**

   The optimisation problem formulated in [6] allows to degrade also high-criticality services to permit to as many low-criticality instances as possible to complete their execution and to not be interrupted. On the other hand, currently mixed-criticality scheduling protocols like the *Bailout Protocol* (BP) solve a different problem since they always degrade low-criticality tasks to allow to higher critical ones to always meet their deadlines. My research has investigated the possibility to integrate the mixed-criticality scheduling protocols based on fixed prioritisation of higher criticality tasks in case of resource shortage with the higher level TRTCM model in which each service has a tolerance interval to exploit in case of failure of some computing resources to keep as many services functioning as possible.

3. **Are the utility functions and tolerance ranges of TRTCM a useful means to increase the amount of tasks scheduled in real-time systems with mixed-criticality services?**

   The TRTCM is based on modeling the system behaviour by means of utility functions. The key idea is that to use a tolerance interval in which performing a graceful degradation of services instead of interrupting them. Such performance decrease is utility driven and each task has a least acceptable service utility under which it should be aborted. Therefore, to

justify the usage of protocols built upon the TRTCM model, there is the need to find and study cases where using the tasks' tolerange ranges allows to increase the amount of tasks that are successfully scheduled, considering first tasks with higher criticality, such that the overall system utility is maximised.

1.4 Contributions

My research has focused on two main directions. On one side, I focused on studying, comparing, evaluating and improving the state-of-the-art of fixed-priority mixed-criticality scheduling protocols on uniprocessor platforms to increase the amount of LO task instances that successfully complete their execution. Subsequently, I further refined the TRTCM model to allow the integration with the underlying scheduling protocols to maximise the overall system utility. My contributions are listed below:

- Introducing the *Lazy Bailout Protocol* (LBP), a BP refinement that allows for a temporary resource denial to LO tasks in case of resource shortage. The key principle of LBP is instead of immediately abandoning LO instances after a HI instance overruns its optimistic WCET estimate, to put them in a low-priority queue for background execution during the system idle instants. The initial BP protocol has subsequently been extended with integration of two complementary techniques that allow to increase the amount of LO instances successfully processed without affecting the schedulability of HI tasks. As a result, I also further refined LBP with such additional techniques and this led to devise the LBPG, LBPS and LBPSG protocols.

- I introduced the *Soft Lazy Bailout Protocol* (SLBP), an LBP variant in which LO tasks have soft deadlines, i.e., they are allowed to have a tardy completion. Like for LBP, I have also integrated the SLBP with the additional strategies that exploit the CPU spare capacity in order to increase the amount of instances scheduled. This led to devise the SLBPG, SLBPS and SLBPSG protocols.

- Definition of a formal criterion to compare different mixed-criticality scheduling protocols on uniprocessor platforms in hard real-time settings with priority given to schedulability of HI tasks. This criterion represents a way to check the impact a particular mixed-criticality scheduling protocol

has within a system. It measures the quality and quantity of the completion rate of task instances generated per scheduling method by comparing first the amount of HI and then that of LO instances scheduled within their deadlines.

- Introducing the ATMP protocol, a framework to integrate the short-term mixed-criticality scheduling protocols with the TRTCM optimisation method to maximise the overall system utility in case of resource shortages via smooth service degradation.

- Comparing and analysing the lazy bailout protocols with the state-of-the-art mixed-criticality fixed-priority scheduling method on single-core architectures. In particular, I compared LBP and SLBP with the existing BP protocols in a hard real-time settings. Furthermore, I also compared and evaluated LBP and SLBP in a soft real-time settings.

- Finally, I showed that the ATMP protocol performs better than standard heuristics in which tasks have no tolerance range to exploit. A multi-core architecture is taken as target to make experiments and results showed that the TRTCM optimisation allows to keep allocated more tasks to cores than the approaches in which some tasks allocated are removed when their overall load are above the manageable threshold of the underlying processing element.

1.5 Evaluation of Mixed-Criticality Protocols

A robust protocol for resource allocation should guarantee that system performances degrade gracefully at runtime in case its behaviour does not conform to the model assumed during the a priori verification. In other terms, it should minimise the amount of task instances that are abandoned or aborted in case of resource shortage.

The existence of the assumption that tasks within a system might have different criticality necessarily leads to devise algorithms that, in case of resource shortage, assure the correct completion or allocation of higher criticality task instances while trying to minimise the service degradation for lower criticality tasks.

As a result, the enhancement of the contributions has been evaluated by considering the increase in the amount of lower criticality instances that complete

within their deadlines or that are kept allocated. Firstly, I considered the fixed-priority mixed-criticality scheduling methods on uniprocessor platforms and then the ATMP protocol.

1.6 Publications

The work related to the TRTCM has been published with the two following papers:

- Raimund Kirner, Saverio Iacovelli and Michael Zolda. *Optimised Adaptation of Mixed-Criticality Systems with Periodic Tasks on Uniform Multiprocessors in Case of Faults.* The 18th IEEE International Symposium On Real-Time Computing (ISORC) Workshop. Auckland (New Zealand), April 2015.

- Saverio Iacovelli, Raimund Kirner and Catherine Menon. *ATMP: An Adaptive Tolerance-based Mixed-criticality Protocol for Multi-core Systems* The 13th International Symposium on Industrial Embedded Systems (SIES) 2018. Graz (Austria), June 2018.

The following journal paper about the LBP protocol has been accepted and is under minor revision process:

- Saverio Iacovelli and Raimund Kirner. *A Lazy Bailout Approach for Dual-Criticality Systems on Uniprocessor Platforms.* MDPI Designs Journal.

1.7 Structure of the Dissertation

The remainder of this dissertation is structured as follows:

- **Chapter 2** provides background information on standard real-time scheduling theory both on uniprocessor and on multiprocessor systems. First, I provide an introductory description on the main concepts involved in real-time systems development. Particular attention is given to the WCET problem, difference between hard and soft real-time systems, schedulability and sustainability. Then, there is a description about the main scheduling approaches on uniprocessor platforms such as scheduling with static and dynamic priority assignments or the usage of utility functions. I also describe two techniques to integrate within the scheduling algorithms to increase the number of task instances completed within their

deadlines. Finally, I provide a detailed review on scheduling on multicore architectures.

- **Chapter 3** introduces the motivations and the context in which the mixed-criticality scheduling has been developed. It describes the limitations of traditional priority-based scheduling algorithms in case of resource shortages and the motivation for the need of novel robust scheduling protocols. Then, this chapter summarises the work produced on mixed-criticality scheduling both on uniprocessor and multiprocessor architectures. Finally, it describes how a utility-based approach, the TRTCM, was designed to include the indication of criticality to optimise specific performance parameters in case of resource shortages.

- **Chapter 4** introduces the LBP protocol, a fixed-priority method to schedule mixed-criticality task sets on single core architectures. It contains the system and task model used for such protocol together with a detailed explanations of its functioning in its execution modes. This chapter also introduces the *Soft Lazy Bailout Protocol* (SLBP), a LBP variant in which low criticality task instances overunning their optimistic WCET estimates are treated as soft real-time tasks. It is also provided an example of how such LBP variant works.

- **Chapter 5** introduces formal predicates to assess and compare performances among different mixed-criticality protocols in hard real-time settings. Such formal predicates and criteria aid to evaluate in which case a mixed-criticality scheduling algorithm outperforms another one. Furthermore, it also contains formal comparisons among different mixed-criticality protocols.

- **Chapter 6** introduces the *Adaptive Tolerance-based Mixed-criticality Protocol* (ATMP), a novel criticality and utility aware partitioning and allocation algorithm. Such method is built upon the TRTCM model and permits to integrate the higher level mixed-criticality optimisation with the underlying mixed-criticality scheduling. This chapter describes the task model used for TRTCM, the partitioning and allocation of tasks to cores and the reconfiguration at runtime. Finally, there is a discussion about the advantages deriving from using ATMP with respect to the safety perspective.

- **Chapter 7** contains the results of experiments made with the LBP approaches and ATMP protocols. This chapter is divided in two sections.

Section 7.1 describes the model, the experiments and the outcome with the scheduling protocols on uniprocessor platforms. Section 7.2 describes and evaluates the results of comparison between the ATMP protocol with a standard policy for allocation and reconfiguration of mixed-criticality task sets on multiprocessor platforms.

- **Chapter 8** concludes the dissertation. It contains a brief summary of each chapter and also the potential application areas of my research. Finally, a outlook about future work and perspective is provided.

1.8 Chapter Summary

In this chapter, I provided an overview about my dissertation and an introductory description of my contributions.

The first sections introduce the topic that is studied in the next chapters together with the motivations that make this field relevant for research and the main challenges to face. Then, Section 1.3 describes my research question and Section 1.4 lists my contributions. Finally, Section 1.7 contains the structure of the dissertation.

Chapter 2

Background

This chapter contains the basic knowledges about real-time scheduling both on single and multi-core architectures. Most of them is recalled afterwards throughout this dissertation.

First of all, Section 2.1 introduces the preliminary knowledges about real-time scheduling, e.g., the WCET estimation, the schedulability analysis and the processor utilisation factor. Section 2.2 introduces the main fixed and dynamic priority strategies used in single-core architecture. Then, it also describes the additional scheduling techniques usually used both in hard and soft real-time systems. Finally, Section 2.3 describes the main scheduling approaches utilised in multiprocessor platforms.

2.1 Real-Time Systems

A real-time system can be defined as a computing system in which computational activities must be performed within predefined timing constraints. A system is real-time if the correctness of its behaviour depends not only on the computational results but also on the timeliness of the computed action, i.e., the correct value must be computed at the right time. In many real-time applications, the recurrent activities represent the major computational demand in the system. Such activities typically arise from sensory data acquisition, low-level servoing, control loops, action planning and system monitoring and need to be cyclically executed at specific rates which can be derived from the applications requirements. When a control application consists of such several concurrent tasks with individual timing constraints, the operating system has to guarantee that each recurrent instance is regularly activated at its proper rate and is completed within its temporal deadline.

Therefore, the fundamental timing metrics to specify real-time requirements is the deadline that represents the instant at which a result must be produced. A real-time system consists of a set of tasks that could be hard, firm or soft according to the severity (or conversely the usefulness) represented by a completion within or after a technical deadline. If a result of a task instance has utility even if it completes after its deadline, then the deadline is named soft, otherwise it is firm. In case severe or even catastrophic consequences could occur if a firm deadline is overrun, then the deadline is called hard [2]. Hence hard deadlines have always to be met, otherwise a critical failure may occur in the system. On the other hand, if a soft deadline is missed, the system keeps working at a degraded level of performance. To exactly evaluate the performance degradation caused by a soft deadline missed, a value function can be associated with each soft task. As showed in [1], Figure 2.1 represents the utility provided by a result when it is produced within its deadline or after by different types of tasks.

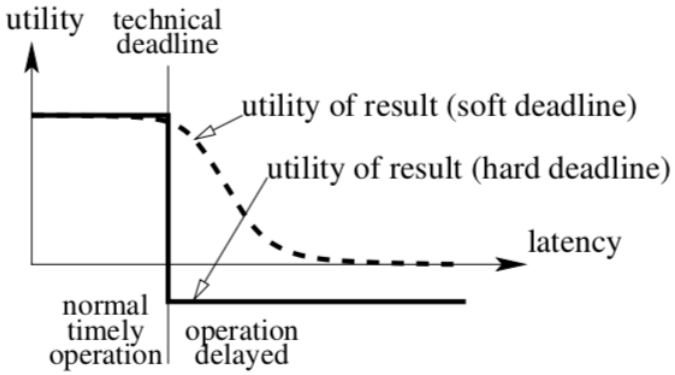


Figure 2.1: Characterisation of utility according to deadline strictness [1]

It is possible to notice as the timing constraint of a hard real-time task is purely deadline-based and a service provides a value to the whole system only if it completes within its technical deadline.

2.1.1 The WCET Estimation

In real-time systems, a prerequisite for the application of any analysis technique to guarantee that each computational activity completes within its deadline is the knowledge about the *Worst-Case Execution Time* (WCET) estimate of all time-critical tasks [2]. In fact, the completion within the technical deadline can only be guaranteed if the WCET of all application tasks and communication actions that are part of the real-time transaction are known a priori. The WCET of a task is a guaranteed upper threshold for the time between task activation and

task termination. Such WCET must be valid for all possible input data and execution scenarios of the task and should be a tight bounds.

Over years, different WCET estimation tools have been developed. The more conservative are typically based upon static analysis of code, which generates WCET bounds that can be very large but that are trustworthy to a very high level of assurance. The analysis of source code identifies the longest path and computing time needed to execute on the specific processor platform. On the other hand, less conservative WCET estimation tools are typically measurement based and tend to obtain smaller estimates but these estimates may be trustworthy to lower levels of assurance since the true worst-case system behaviours may not have become revealed during the measurements. The experimental WCET estimation is done by measuring the maximum execution time of each task over a large amount of input data.

Once all computation times are evaluated, the feasibility of the system can be analysed using several guarantee algorithms proposed in the literature for different scheduling algorithms and task models.

2.1.2 Hard Real-Time Systems

Historically, the real-time computing technology has been primarily developed to support safety-critical systems or industrial systems that have to guarantee certain performance requirements with a limited degree of tolerance. Within the so-called hard real-time systems, most computational activities are characterised by critical timing requirements that have to be met in all operating conditions in order to guarantee the correct system behaviour. In such context a deadline missed is not tolerated, either because it could have catastrophic effects on the controlled environment or because it could jeopardise the guarantee of some stringent performance requirements. In such cases, a task finishing after its deadline is considered not only late, but also wrong, since it could jeopardise the whole system behaviour. A hard real-time system must execute a set of recurrent real-time tasks such that all time-critical tasks meet their specified deadlines. Every task needs computational, data and input/output resources to be processed.

In order to guarantee a predefined performance, hard real-time systems are designed under worst-case scenarios, derived by making pessimistic assumptions on the system behaviour and the environment. In this case, all resources are statically allocated to tasks based on their maximum requirements.

Such systems are often modelled as a set of computational tasks to be exe-

cuted concurrently on the selected hardware platform by a real-time scheduler. The computational tasks are characterised by a WCET and are recurrently activated by input stimuli with a certain activation pattern.

2.1.3 Soft Real-Time Systems

In many cases, although the timeliness in processing and completing recurrent task instances is required because of the sensitivity to delay and jitter, the scheduling methodologies devised for the hard real-time systems are not suited since deadline misses may decrease the *Quality of Service* (QoS) but do not cause critical system faults. Therefore, over years, the real-time systems technology used to develop systems with safety-critical requirements has been extended to support novel application domains characterised by less critical timing requirements, scarce resources and more dynamic behaviour such as multimedia systems, monitoring apparatuses, telecommunication networks, mobile robotics, virtual reality and interactive computer games. In such systems, also called soft real-time systems, task instances missing a deadline do not cause catastrophic consequences on the environment but only a performance degradation, often evaluated through some QoS parameter [23].

The soft real-time systems may allow for some deadline to be missed by no more than a certain amount. In these cases, a late completion could be still useful and better than no completion at all. As an example, tasks within an embedded multimedia player that reproduces a movie at a predefined regular periodic rate could miss some deadlines. This would cause a degradation of the perceived service provided but no catastrophic consequence would happen. However, the fact that a soft real-time application may tolerate a certain degree of performance degradation does not mean that timing constraints can be completely ignored. In general, a certain quality of service level needs to be enforced on the computational tasks to satisfy a desired performance requirement. If too many deadlines are missed, and especially if they are consecutive, then there is no way to keep the system performance above a certain threshold.

Furthermore, real-time systems often could consist of a mixture of hard and soft real-time tasks. Such systems should guarantee that hard real-time tasks meet their deadlines while trying to maximise the completion rate for soft real-time tasks. In such cases, the guarantees for hard tasks are based on their *Worst-Case Execution Time* (WCET) estimates while those relative to soft tasks could be based on the mean execution times of each of their task instances. As an example, in 1998, Abeni and Buttazzo described a server-based mechanism

for scheduling soft multimedia tasks without jeopardizing the a priori guarantee of hard real-time activities [24]. The hard tasks were scheduled by the EDF while each soft task was handled by a dedicated server, the *Constant Bandwidth Server* (CBS), with the aim to minimize their mean execution after the deadline. The CBS server is seen as a hard task providing a desired level of service to soft tasks.

2.1.4 Task Models for Real-Time Scheduling

In real-time systems, the workload is typically characterised as being generated by a finite collection of recurrent tasks or processes. Each task generates a potentially unbounded sequence of jobs. Therefore, the workload is modeled as being comprised of basic units of work known as jobs. Various models have been proposed for representing tasks; some of the more widely used models include the *Liu and Layland model* [25] and the *three-parameter model* [26].

The Liu and Layland task model is the simplest model for representing recurring processes executing upon a shared platform. In this model, a task τ_i is characterised by just an ordered pair of two parameters as follows:

$$\tau_i = \langle P_i, C_i \rangle$$

in which P_i represents both period and deadine of the task and C_i represents the WCET requirement. Such a task generates a potentially infinite sequence of jobs that are invoked at each non-negative integer multiple of P_i. Task invocations are also called job releases or job arrivals. Each invocations requires at most C_i units of processor time and must complete its execution within P_i time units. The first job may arrive at any instant and the arrival times of any two successive jobs are at least P_i time units apart. The task arrival time is the instant when a request for a task execution is made. Depending on the arrival times, it is possible to distinguish between the three different following task types:

1. *periodic task*: a task is periodic if all future arrival times are known a priori by adding multiples of the known period to its initial arrival time.

2. *sporadic task*: a task is sporadic if there is no arrival time known a priori but it is assumed a minimum inter-arrival time between any two request times of such task.

3. *aperiodic task*: a task is defined aperiodic if there is no constraint on the arrival times of task activations.

A collection of periodic tasks is referred to as a periodic task set or periodic task system and is usually denoted as τ. A task system τ consists of a set of tasks as follows:

$$\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$$

If the arrival time of each first task instance in τ occurs at time $t = 0$, then the task set is referred to as *synchronous*, otherwise it is referred to as *asynchronous*.

The three-parameter task model was proposed to represent tasks in which the relative deadline is different from the related period. A three-parameter sporadic task denoted by τ_i is thus represented by the following tuple:

$$\tau_i = \langle P_i, D_i, C_i \rangle$$

Such a task generates a potentially infinite sequence of jobs. The first job may arrive at any instant and the arrival time of two successive jobs are at least P_i time units apart. Each job has a WCET indicated by C_i and a relative deadline that occurs D_i time units after its arrival time. A three-parameter task set consists of a finite number of such three-parameter tasks executing upon a shared platform. By allowing for the specification of a relative deadline parameter in addition to a period, such model offers a means of specifying recurrent workloads that may occur infrequently, i.e., large periods, but that are urgent because of small deadlines. Depending upon the relationship between the value of the relative deadline and period parameters of the tasks in it, a task set could be classified as follows:

1. *implicit deadline task set*: the relative deadline of each task is equal to the task's period:

   $$\forall\ \tau_i \in \tau.\, D_i = P_i$$

   It is worth to notice that the implicit deadline task sets are those expressed by the Liu and Layland model.

2. *constrained deadline task set*: the relative deadline of each task is no larger than the task's period:

   $$\forall\ \tau_i \in \tau.\, D_i \leq P_i$$

3. *arbitrary deadline task set*: each task can have a deadline that is smaller, equal or even larger than its period:

   $$\forall\ \tau_i \in \tau.\, D_i \leq P_i \vee D_i > P_i$$

Furthermore, given that a job j_i with deadline D_i completes at time F_i, the timing parameter that usually describes and quantifies the relationship between completion time and deadline in a job is the *lateness*, formally represented as below:

$$lateness_i = F_i - D_i \tag{2.1}$$

The lateness is the algebraic summation between the finishing time and deadline. Note that if a task completes before its deadline, its lateness is negative. Moreover, the formula 2.1 allows to obtain two metrics usually used to measure performances in soft real-time systems, i.e., the exceeding time and the amount of time left from the job completion to its deadline:

$$tardiness_i = max(0,\ lateness_i) \tag{2.2}$$

$$earliness_i = max(0,\ -lateness_i) \tag{2.3}$$

In particular, the tardiness represents the completion delay of a task with respect to its deadline and is useful because represents the time a task instance stays active after its deadline. It is worth to notice that hard real-time tasks must have tardiness equal to zero while for soft real-time tasks it is important that the tardiness is usually reasonably bounded or minimised.

2.1.5 Schedulability Analysis

The scheduling theory is concerned with the efficient allocation of computational resources, which may be available in limited amounts, among competing demands in order to optimise specified objectives. In particular, the real-time scheduling theory deals with resource allocation in real-time computer systems.

The timing constraints within which results must be provided are expressed as deadlines and thus the schedulability of a task set implies that all deadlines are satisfied if the system behaves according to its parameterized specification. A task set is schedulable with respect to a specified scheduling policy if all jobs produced by it will complete within their deadline when executed on its target platform with that scheduling policy.

To guarantee that all tasks within a system will be successfully scheduled, it is necessary to apply a test that is appropriate for the dispatching policy of the execution platform. A test that determines whether a set of tasks can be scheduled such that each task instance will meet its deadline is called a schedu-

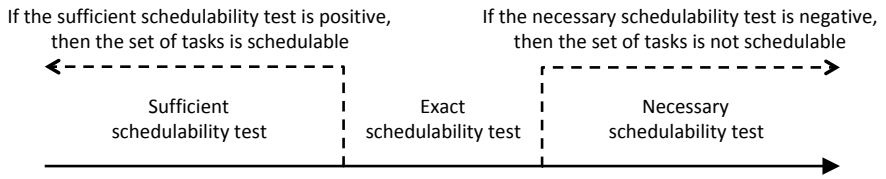


Figure 2.2: Sufficient, necessary and exact schedulability tests [2]

lability test. It is possible to distinguish between exact, necessary and sufficient schedulability tests.

In particular, a schedulability test for some specific scheduling algorithm is referred to as sufficient, if all the task sets and priority orderings that are deemed schedulable according to the test are in fact schedulable under the scheduling algorithm. Similarly, a schedulability test is referred to as necessary, if all the task sets and priority orderings that are deemed unschedulable according to the test are in fact unschedulable under the scheduling algorithm. A schedulability test that is both sufficient and necessary is referred to as exact [27]. Clearly sufficiency is critically important for most hard real-time systems. Figure 2.2 describes the relationship between the three types of tests explained above [2].

The schedulability analysis must be performed offline; in order to do so, parameters characterizing the run-time workload such as WCET and period must be estimated prior to run-time. The system designer performs an offline analysis to guarantee that the system is able to achieve a minimum desired performance in all operating conditions that have been predicted in advance.

Schedulability tests play an important role in the verification of safety-critical real-time systems in which a deadline missed may lead to catastrophic consequences. Hence, within hard real-time systems, the schedulability test is a instrument to guarantee that every deadlines will always be met. However, a guarantee test based on worst-case scenarios could lead to a significant waste of resources, especially in systems characterised by a highly dynamic behaviour. Therefore, the consequence of such worst-case design methodology is that high predictability is achieved at the price of a very low efficiency in resource utilisation and of an increase of the overall system cost. Such a waste of resources could be justified for critical applications in which a single deadline miss may cause catastrophic consequences but it does not represent a good solution for applications in which several deadline misses can be tolerated as long as a average jobs completion is guaranteed offline.

In soft real-time systems, it is required to estimate the number and frequency

of missed deadlines. In this case, the schedulability test can give useful indications about the number of deadlines missed or about the maximum extent of execution after the deadline has expired. This problem is addressed by adopting a probabilistic framework to rigorously characterise a soft real-time system. Hence, the analysis for hard real-time systems is extended to cope with statistically distributed execution and/or interarrival times. The aim is that to perform a probabilistic schedulability analysis of real-time task sets to provide a relaxed form of guarantee for systems with highly variable execution behaviour. The objective of the analysis is to derive a probability for each task to meet its deadline or, in general, to complete its execution time within a given interval of time.

2.1.6 Sustainable Schedulability Test

The notion of sustainability was introduced in 2006 by Baruah and Burns to formalise the expectation that a system that is schedulable under its worst-case specifications should remain schedulable when one or more properties of at least one task in a task set (e.g., inter-arrival time or execution time) are less pessimistic than predicted [28]. Baruah argues that a sufficient and sustainable schedulability test is more important than a sufficient and necessary one and that, generally, it should be a good engineering practice to use sustainable tests if possible and classify common uniprocessor schedulability tests according to whether they are sustainable.

A scheduling algorithm is said to be sustainable with respect to a task model, if and only if schedulability of any task set compliant with the model implies schedulability of the same task set modified by decreasing execution times, increasing periods or inter-arrival times and increasing deadlines. Similarly, a schedulability test is defined to be sustainable if any task set deemed schedulable by the test remains so if it behaves better than mandated by its system specification. More precisely, a schedulability test for a scheduling policy is sustainable if any task set deemed schedulable by the schedulability test remains schedulable when the parameters of one or more individual jobs are changed in decreased execution requirements, later arrival times, smaller jitter and larger relative deadlines. A schedulability test may be sustainable with respect to some, but not all, task parameters. Baruah and Burns showed that all sufficient schedulability tests for fixed-priority preemptive scheduling are sustainable with respect to execution time requirement but no exact schedulability test for the fixed-priority preemptive scheduling of periodic task systems can be sustainable with respect to jitter.

It is worth to notice that declaring a schedulability test to be sustainable represents a stronger claim than simply declaring that a set of tasks deemed schedulable by the test would remain schedulable with better parameters (e.g., with larger periods or relative deadlines or with smaller execution time or jitters) since a sustainable system must continue to meet all deadlines even if the parameters change occurs at run-time and such parameters could change back and forth arbitrarily many times.

Intuitively, the sustainability requires that schedulability be preserved in situations in which it should be easier to ensure the same task set being feasible. From this point of view, sustainability is the opposite property of robustness. A robust system retains schedulability even when it operates beyond the worst-case assumptions used in its schedulability test, e.g., when jobs arrive earlier than expected or have greater execution requirement than permitted. Clearly a system can never be fully robust since at some point the system will become so overloaded that it will fail. However, it is not ruled out that a system could in principle be fully sustainable because no amount of under-load need forces failure.

2.1.7 Processor Utilisation Factor

For each task τ_i within a task set, it is possible to compute the amount of time such task uses the processor. As a consequence, the utilisation of a task set is defined as sum of the utilisations of all tasks in it. Therefore, given a set of n periodic tasks, the *processor utilisation factor* U is the fraction of processor time spent in the execution of the task set [25]. The utilization factor of each individual task τ_i denotes the percentage of time such task requires service from the processor. Since C_i/P_i is the fraction of processor spent executing task τ_i, the utilisation factor for n tasks is given by:

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i} \tag{2.4}$$

The processor utilisation factor provides a measure of the computational load on the CPU due to the periodic task set. Although the CPU utilisation can be improved by increasing tasks' computation times or by decreasing their periods, there exists a maximum value of U below which τ is schedulable and above which τ is not schedulable anymore. Such a limit depends on the task set and on the algorithm used to schedule the tasks. Let $U_{ub}(\tau, A)$ be the upper bound of the processor utilisation factor for a task set τ under a given algorithm A. When

$U = U_{ub}(\tau, A)$, the set τ is said to fully utilise the processor. In this situation, τ is schedulable by A but an increase in the computation time in any of the tasks will make the set infeasible. For a given algorithm A, the *least upper bound* $U_{lub}(A)$ of the processor utilisation factor is the minimum of the utilisation factors over all task sets that fully utilise the processor:

$$U_{lub}(A) = min_{\tau}\ U_{ub}(\tau, A) \tag{2.5}$$

U_{lub} defines an important characteristic of a scheduling algorithm useful for easily verifying the schedulability of a task set. In fact, any task set whose processor utilisation factor is less than or equal to this threshold is schedulable by the algorithm. Lastly, if the utilisation factor of a task set is greater than 1.0, the task set cannot be scheduled by any algorithm.

2.2 Scheduling on Uniprocessor Systems

2.2.1 Preemption and Priority Assignments

Real-time scheduling algorithms can be classified according to different criteria. One criterion might consider when preempt and interrupt a job that is executing to allocate computing resources to the next job. Once established that an instance running on a CPU can be preempted and then resumed later at runtime. Another categorisation could consider the way the priorities are assigned to jobs.

According to the dispatching policy, the scheduling algorithms could be distinguished as follows:

- *Non-preemptive*: once a job begins execution, it continues to execute until it has completed. The preemption is totally forbidden and if a job with stricter timing requirements needs computing resources, it will wait that the current one terminates.

- *Fully preemptive*: the current executing job can be interrupted by the scheduler whenever a more urgent one requests service to be resumed at a later point in time.

- *With deferred preemption*: the preemption is allowed only at specified times and points during the execution. Such systems are also referred to as systems with limited preemption.

Within systems in which preemption is allowed, it is typically assumed that the cost for preemption and context switch of jobs is zero.

The scheduling algorithms could also be differentiated according to how they assign priority and according to the complexity of the priority scheme as below:

1. *Static priority*: a unique priority is associated with each task and all jobs generated by a task have priority associated with that task. Thus, if task τ_1 has higher priority than task τ_2, then whenever both have active jobs, the τ_1's job will have higher priority over τ_2's job. An example of a scheduling algorithm in this class is the RM algorithm [25].

2. *Dynamic priority*: an instance generated by a task can have a priority that is sometime higher and other times smaller than an instance generated by a second task. Examples of such category are the *Earliest Deadline First* (EDF) [25], for which the priority is given by the absolute deadlines, and the *Least Laxity First* (LLF) [3], for which the job priority is computed according to its current laxity.

Figure 2.3 shows the main static and dynamic priority assignment schemes and algorithms.

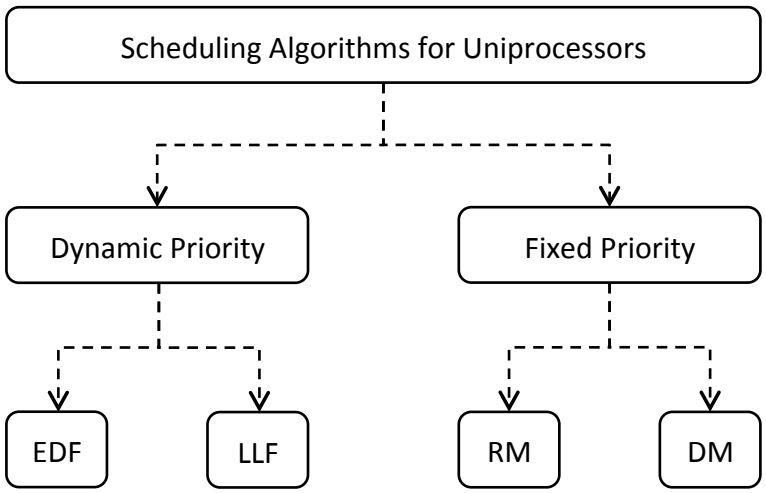


Figure 2.3: Types of scheduling algorithms [3]

2.2.2 Fixed Priority Scheduling

The *Rate Monotonic* (RM) scheduling is a simple rule that assigns priorities to tasks according to their request rates. Specifically, tasks with higher request rates, i.e., with shorter periods, will have higher priorities. Since periods are constant, RM is a fixed priority assignment: a priority is assigned to the task

before execution and does not change over time. Moreover, RM is intrinsecally preemptive: the currently executing job is preempted by a newly arrived job with shorter period. In 1973, Liu and Layland showed that RM is an optimal priority assignment policy for synchronous periodic or sporadic task sets with implicit deadlines in the sense that no other fixed-priority algorithms can schedule a task set that cannot be scheduled by RM [25]. Liu and Layland also derived the least upper bound of the processor utilisation factor for a generic set of n periodic tasks. Therefore, for an arbitrary set of periodic tasks, the least upper bound of the processor utilisation factor under the RM scheduling algorithm is

$$U_{lub} = n(2^{1/n} - 1) \tag{2.6}$$

A sufficient condition for the RM schedulability is the following:

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(2^{1/n} - 1) \tag{2.7}$$

However, Kuo and Mok provided a potentially superior utilisation bounds for task systems in which the task period parameters are harmonically related. Let $\tilde{n}$ denote the number of *harmonic chains* in the task set, then a sufficient condition for such task set to be RM-schedulable is that:

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq \tilde{n}(2^{1/\tilde{n}} - 1) \tag{2.8}$$

Such a result was also later confirmed by Buttazzo [29].

The *Deadline Monotonic* (DM) priority assignment weakens the *period equals deadline* constraint within a static priority scheduling scheme. This algorithm was first proposed in 1982 by Leung and Whitehead as an extension of RM, where tasks can have relative deadlines less than or equal to their period (i.e., constrained deadlines) [30]. The DM priority assignment is optimal for task sets with constrained deadlines [30]. However, DM is not optimal for task sets with arbitrary deadlines [31] or for asynchronous periodic task sets. According to DM algorithm, each task is assigned a fixed priority inversely proportional to its relative deadline. Thus, at any instant, the task with the shortest relative deadline is executed. Since relative deadlines are constant, DM is a static priority assignment. As RM, DM is normally used in fully preemptive mode: the currently executing job is preempted by a newly arrived job with shorter relative deadline. The feasibility of a task set with constrained deadlines could be guaranteed using

the utilisation based test, by reducing tasks' periods to relative deadlines:

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq n(2^{1/n} - 1) \tag{2.9}$$

However, such a test would be quite pessimistic, since the workload on the processor would be overestimated. A less pessimistic schedulability test can be derived by noting that

- the worst-case processor demand occurs when all tasks are released simultaneously, i.e., at their critical instants;
- for each task τ_i, the sum of its processing time and the interference imposed by higher priority tasks must be less than or equal to D_i;

To find a sufficient and necessary schedulability test for DM, the exact interleaving of higher priority tasks must be evaluated for each process. In general, this procedure is quite costly since, for each task τ_i, it requires the construction of the schedule until D_i. Audsley et al. proposed [32, 33] an efficient method for evaluating the exact interference on periodic tasks and derived a sufficient and necessary schedulability test for DM, called *Response Time Analysis* (RTA). According to the method proposed by Audsley et al. the longest response time R_i of a periodic task τ_i is computed, at the critical instant, as the sum of its computation time and the interference I_i of the higher priority tasks:

$$R_i = C_i + I_i \tag{2.10}$$

where

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{P_j} \right\rceil C_j \tag{2.11}$$

Hence,

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{P_j} \right\rceil C_j \tag{2.12}$$

A further necessary and sufficient test for checking the schedulability of constrained deadline task sets in fixed priority systems is named *Workload Analysis* (WA) and was proposed by Lehoczky, Sha, and Ding [34]. The test is based on the concept of *Level-i Workload* $W_i(t)$. The Level-i workload $W_i(t)$ is the cumulative computation time requested in the interval $(0, t]$ by task τ_i and by all the tasks with higher priority.

The Level-i workload for a set of synchronous periodic tasks can be computed as follows:

$$W_i(t) = C_i + \sum_{h=1}^{i-1} \left\lceil \frac{t_i}{P_h} \right\rceil C_h \tag{2.13}$$

A set of fully preemptive periodic tasks τ can be scheduled by a fixed priority algorithm if and only if

$$\forall i = 1, \ldots, n \; \exists t \in (0,\ D].\, W_i(t) \leq t$$

2.2.3 Dynamic Priority Scheduling

The *Earliest Deadline First* (EDF) algorithm is a dynamic scheduling rule that selects task instances according to their absolute deadlines. Specifically, tasks with earlier absolute deadlines will be executed at higher priorities. Therefore, EDF is a dynamic priority assignment. Moreover, it is typically executed in preemptive mode, thus the currently executing job is preempted whenever another periodic instance with smaller deadline becomes active. It is necessary to notice that EDF does not make any specific assumption on the periodicity of the tasks; hence, it can be used for scheduling periodic as well as aperiodic tasks. EDF is an optimal scheduling algorithm for sporadic task sets regardless of the deadline constraints [35]. The schedulability of a periodic task set handled by EDF can be verified through the processor utilisation factor. In this case, however, the least upper bound is one; therefore, tasks may utilise the processor up to 100% and still be schedulable. Therefore, a set of periodic tasks is schedulable with EDF if and only if

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1 \tag{2.14}$$

2.2.4 Scheduling with Utility Functions

There have been approaches to extend the pure deadline-based real-time specification by a generic function to measure or optimise the system utility. As an example, to overcome the difficulty with deadlines in radar scheduling problems, Jensen et al. proposed a *real-time driven scheduling model* [36]. The time-driven scheduling was the first attempt to extend the classical model and to provide a tool to enhance the effectiveness of real-time scheduling by exploiting the fact

that the job completion has a value provided to the system which can be expressed as a function of time and such value could be positive also after the chosen deadline [36]. The authors simulated a real-time system in which each task instance has a value for completing at any time and then the system is rewarded with the value determined by that function when the instance terminates. The sum of the resulting values for all tasks to be performed provides the metrics for determining the system performance according to each scheduling algorithms.

The research about value functions for adaptive resource management in dynamic time-critical systems has been afterwards formalised as TUF/UA paradigm [37, 38]. A *Time/Utility Function* (TUF) generalises the standard deadline constraint since it specifies the utility provided to the system by the completion of an action at a specified time. Then, the optimality criterion is specified by the collective value provided by each task as summed utilities and it is named *Utility Accrual* (UA). From this point of view, the classical deadline can be represented as a binary-valued step function that has full utility till the chosen deadline and no utility after. In general, a TUF is application dependent and adds more expressiveness to specify the semantics of soft time constraints.

Then, such approach has been extended and developed to optimise parameters such as energy consumption that is critical in mobile and portable, battery-powered systems and to improve memory management in embedded systems. The utility accrual via time-utility functions overcomes the shortcomings of standard deadline-based real-time scheduling whenever a late completion has still a decreased but still acceptable utility for the overall system correctness.

The TUF/UA paradigm adds more flexibility to the pure deadline-based scheduling since it allows to evaluate the utility of the output for any latency and makes possible to express also negative utility. On the other hand, it does not remove nor change the basic limitation, i.e., the critical latency after which utility is not positive anymore and the assigned technical deadline are not explicitly specified nor distinguished. This still represents a mismatch with the engineering practices. The first usage of utility functions to practically specifying how to overcome the limits of traditional deadline-based scheduling is due to Kirner [22, 1]. His work points out that real-time services have a technical deadline chosen by system engineers as maximum latency to provide the output that differs from the critical latency imposed by the environment after which the service utility becomes zero or even negative. Therefore, the standard real-time model represents a simplification since the deadline is used to determine both the critical latency and the technical deadline used as design parameter.

Kirner proposed a further refinement called *Tolerance-based Real-Time Computing Model* (TRTCM) that makes explicit the difference between the latency where the resulting utility could become zero and the latency chosen as technical deadline.

2.2.5 Complementary Scheduling Techniques

Adaptive and dynamic behaviour is seen as one of the key characteristics of the modern real-time systems. The fixed priority preemptive scheduling is usually used in such real-time systems but it is inflexible in its purest form. Provided that in a task set deemed feasible by schedulability analysis all hard jobs must always complete within their deadlines, it is inevitable that the processor and other resources will be under-utilised at run-time. This occurs for many reasons, including jobs not taking worst case execution paths, sporadic jobs not arriving at their maximum rate and hardware speed-ups such as caching and pipelining, which could not be predicted by worst-case execution time analysis. Furthermore, generally, with modern processors it is becoming increasingly difficult to produce tight upper bounds on the worst-case execution times of real-time tasks without incorporating excessive pessimism [39].

The resources not required at run-time are usually termed as spare capacity [40]. Such spare resources instead could be profitably exploited by other jobs, either hard or soft. Davis has classified the spare capacity within a real-time system into the three following groups [41]:

- *Extra capacity*: it is the capacity which is not allocated for real-time tasks during the design phase. This can be identified off-line.

- *Gain time*: it is the processor time guaranteed to a task off-line but not required at run-time. It is produced when the real-time task instances execute in less than their worst-case execution time estimations. This may only be reclaimed at run-time since it depends on the actual executions of tasks [41].

- *Spare time*: it is the capacity produced in situations in which sporadic tasks do not arrive at their maximum rate.

Most flexible scheduling algorithms are mainly focused on reclaiming the extra capacity of the system, usually called *slack time*. Only a few research approaches have discussed how to reclaim gain time [40, 42, 43]. Both strategies

allow to increase the amount of jobs scheduled. However, the former approaches are performed offline while the latter check if there is spare capacity at run-time.

Schedulability tests provide no indication of the extent to which the WCET estimates of tasks of feasible systems may be increased without causing deadlines to be missed. Punnekkat et al. provided a general approach to the sensitivity analysis of task sets regardless of the priority assignment algorithm that is used [44]. In this domain, sensitivity analysis refers to the study of the permissible changes of temporal task characteristics which still lead to a feasible task set. Such approach aids system developers in incorporating changes to the system while ensuring that the schedulabitity guarantees remain intact. Most of metrics consider changes in the WCETs [45].

With regard to this, it is important the definition of *critical scaling factor* α^*. In particular, given a task set in which it is possible to identify spare CPU capacity prior to runtime and where each task τ_i could be represented as below:

$$\tau_i = \langle P_i, C_i \rangle$$

then the *critical scaling factor* α^* is the largest possible factor for each task's worst case execution time C_i above which some task instance will miss its deadline at the critical instant phasing. Conversely the task set remains schedulable for all $\alpha \leq \alpha^*$. Katcher et al. utilise the concept of scaling factor α^* to increase the utilisation factor of each task within a task set as follow [46]:

$$\forall \tau_i \in \tau . \sum_{i=1}^{n} \alpha^* \cdot \frac{C_i}{P_i}$$

However, to increase the overall system load till the maximum point at which it remains schedulable, it is possible also to consider the maximum permissible change in the WCET of just a single task or of one module contained in one or more tasks.

On the other hand, the online method reclaims the gain time collection at runtime. The gain time is defined by noting that an invocation of task τ_i will produce a job j_i that very likely will have an execution time $et(j_i)$ smaller than the WCET. Therefore, the gain time refers to the difference between the execution time actually used by a job and the execution time budget that was allocated. The most important property of any scheme for exploiting the gain time is that the schedulability of hard tasks must not be affected. A number of mechanisms exist that can make this gain time available for usage by other jobs without affecting the schedulability. The gain time is defined as follows:

$$g_i = C_i \ - \ et(j_i)$$

where $et(j_i)$ is the execution time of job j_i and C_i is the related assigned time threshold. At runtime, it is likely that many jobs will complete in less than their optimistic execution time threshold estimates. In a fixed priority scheme such unused resource will become available to background or lower priority tasks. The gain time g_i is added to the execution time budget of the next lower priority active job, i.e., the next job in the ready queue. It is worth to note that g_i can never be negative. Passing the gain time from one job to another makes less likely that jobs requiring more execution time than expected will actually exceed their execution time budgets.

2.2.6 Techniques for Soft Real-Time Systems

To provide appropriate support to soft real-time systems, new methodologies have been investigated to achieve more flexibility in handling task sets with dynamic behaviour as well as higher efficiency in resource exploitation. An offline analysis of the system is required to estimate the number and frequency of missed deadlines. It is necessary to check at runtime the amount of deadlines missed. Typically, there are three types of requirements for soft real-time tasks [23]:

1. Bounds on the number of deadline misses in an interval of time. It is usually required that at most m job deadlines over n instances could be missed.

2. Bounds on the tardiness of a task. Task's instances can complete after their deadlines, but the delay must be bounded.

3. Having probabilistic bounds, e.g., it is possible to look for upper bounds on the probability of having a deadline missed. From this point of view, a probabilistic analysis needs a probabilistic characterisation of the execution time of a task.

Another way to address the system's unpredictability is to use some kind of feedback to dynamically adapt the scheduler behaviour so that some selected QoS metric is kept under control even in the presence of overload situations. Since, it is not possible to prevent overloads but it is possible to minimise their effects, the feedback mechanism creates a reactive system. The feedback mechanism uses information about the actual behaviour of a scheduling system to dynamically adapt the scheduling algorithms such that the intended behavior is

achieved [2]. The feedback scheduling is based on the observation of relevant performance parameters of the scheduling system. As a result, by looking at the scheduling and control problems in an integrated fashion, better overall results can be achieved in many control scenarios. To apply feedback techniques to real-time scheduling, it is necessary first to select a QoS index to control, the so-called feedback variable, and then to select a scheduling parameter to be adapted. For example, the arrival rate of a task can be dynamically adjusted based on the observed average of the deadlines missed.

2.3 Scheduling on Multiprocessor Systems

Since the beginning of this century the computer chip market has experienced what has been named as multicore revolution, that is pushing all major chip producers to switch from single to multicore platforms. Companies building embedded real-time systems are driven by a profit motive. To succeed, they aim to meet the needs and desires of their customers by providing systems that are more capable, flexible and effective than their competitors and by bringing these systems to market earlier. This desire for technological progress has resulted in a rapid increase in both software complexity and the processing demands placed on the underlying hardware. To address demands for increasing processor performance there is now an increasing trend towards using multiprocessor platforms for high-end real-time applications. As a result, motivated by the vastly increased computational demand of real-time workloads and by the trend in hardware toward multicore and multiprocessor CPUs, real-time systems are increasingly coming to be implemented upon multiprocessor platforms. A large body of research has been performed addressing the various issues, challenges and opportunities arising from this move towards multiprocessor platforms [47]. The multiprocessor real-time scheduling theory is concerned with the development of techniques and methodologies that enable the correct and resource-efficient implementation of real-time systems upon multiprocessor platforms.

2.3.1 Classification of Multiprocessor Systems

In multiprocessor computing platforms there are several processors available upon which jobs may execute. To completely specify a multiprocessor platform it is necessary to know various details, e.g., the number of processors that comprise the platform, the computing capabilities of such processors and wheter they are connected to each other. In addition, it is necessary to know wheter the

platform supports preemption and inter-processor migration.

Scheduling theorists distinguish between at least three different kinds of multiprocessor machines (identical, uniform or unrelated) according to the relative computing capabilities of the different processors:

- *Identical parallel machines*: these are multiprocessors in which all the processors are identical, in the sense that they have the same computing power. Hence the rate of execution of all tasks is the same on all processors.

- *Uniform parallel machines*: each processor in a uniform (or related) parallel machine is characterised by its own computing capacity, with the interpretation that a job that executes on a processor of computing capacity s for t time units completes in $s \times t$ units of execution. In this case, the rate of execution of a task depends only on the speed of the processor. Thus a processor of speed 2 will execute all tasks at exactly twice the rate of a processor of speed 1. It is worth to notice that identical parallel machines are a special case of uniform parallel machines, in which the computing capacities of all processors are equal.

- *Unrelated parallel machines*: in such machines, there is an execution rate $r_{i,j}$ associated with each job-processor ordered pair $\langle j_i, c_j \rangle$ with the interpretation that job j_i completes in $r_{i,j} \times t$ units of execution by executing on core c_j for t time units. Hence, in an unrelated multiprocessor a different execution rate may be specified for each job upon each processor. In this case, since processors are different, the rate of execution of a task depends on both the processor and the task. Furthermore, not all tasks may be able to execute on all processors.

Observe that identical multiprocessor are a special case of uniform multiprocessors and uniform multiprocessors are a special case of unrelated multiprocessors. Much of multiprocessor real-time theory has focused upon identical multiprocessor platforms. However, there is an increasing trend in industry towards heterogeneous multicore CPUs containing specialised processing elements such as *Digital Signal Processing* (DSP) cores, *Graphics Processing Units* (GPUs), in addition to general-purpose processing cores.

2.3.2 Taxonomy of Multiprocessor Scheduling Algorithms

Given a set of tasks to be processed, the multiprocessor scheduling can be viewed as attempting to solve two problems:

- *Allocation*: on which processor a task should execute.
- *Priority assignment*: when and in what order each job should execute with respect to other jobs.

In designing scheduling algorithms for multiprocessor environments, it is possible to distinguish between two main approaches: partitioned and global. In partitioned scheduling, each task is allocated to a processor and no task nor job migration is permitted. Conversely, the global scheduling permits both task level migration (i.e., different jobs of a task may execute on different processors but each job can only execute on a single processor) as well as job level migration (i.e., an individual job that is preempted may resume execution upon a processor different from the one upon which it had been executing prior to preemption). Finally, there are also some hybrid class of algorithms that combine characteristics of both previous approaches. Figure 2.4 summarises the existing approaches to multiprocessor scheduling.

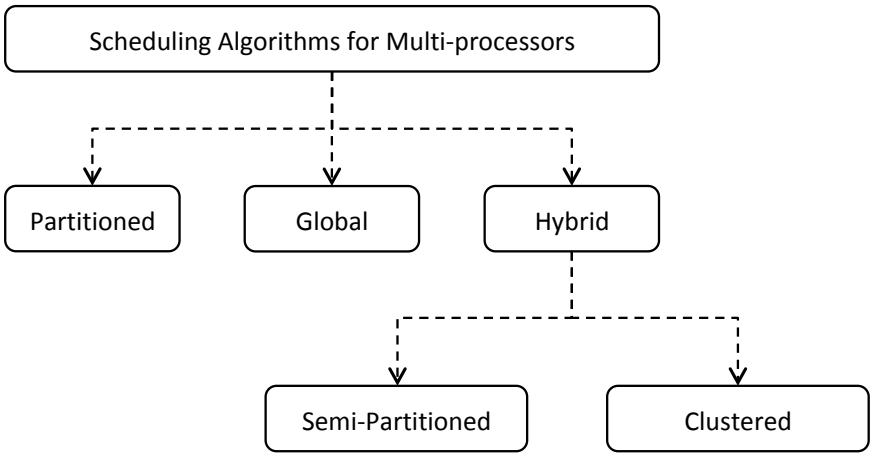


Figure 2.4: Types of scheduling algorithms [3]

The majority of research into global scheduling algorithms has focussed on models where arbitrary migration (job-level migration) is permitted. It has been proven by Leung and Whitehead that the partitioned and global approaches to static-priority scheduling on identical multiprocessors are incomparable [30], in the sense that:

1. there are task sets that are feasible on m identical processors under the partitioned approach but for which no priority assignment exists which would cause all jobs of all tasks to meet their deadlines under global scheduling on the same m processors.

2. there are task sets that are feasible on m identical processors under the global approach but which cannot be partitioned into m distinct subsets such that each individual partition is feasible under a static priority policy on a single core.

Such a result provides a very strong motivation to study both the partitioned and the non-partitioned approaches to static-priority multiprocessor (identical as well as uniform) scheduling, since it is provably true that neither approach is strictly better than the other. However, in 2006, Baker made an empirical comparison between the global and partitioned EDF scheduling algorithms available [48]. The empirical performance metrics used to compare the two approaches was the number of randomly generated task sets that were schedulable according to each algorithm. The conclusion of this study was that although the two approaches are incomparable, the partitioned approach appeared to outperform the global approach on this metric by a significant margin.

A multiprocessor scheduling algorithm is said to be work conserving, if the algorithm never idles a processor while there is some active job awaiting execution which may legally execute upon this processor [49]. Unlike global scheduling, the partitioned scheduling algorithms are not work-conserving, as a processor may become idle but cannot be used by ready tasks allocated to a different processor.

With regard to sustainability properties, while EDF and fixed priority scheduling are sustainable algorithms with respect to uniprocessor scheduling for both synchronous periodic and sporadic tasksets, the same is not true of global EDF and global fixed task priority multiprocessor scheduling. The sustainability of schedulability tests for global EDF has been investigated by Baker and Baruah [50].

2.3.3 Partitioned Scheduling

Partitioned scheduling has the following advantages compared to global scheduling:

1. If a task overruns its worst-case execution time budget, then it can only affect other tasks on the same processor.

2. As each task only runs on a single processor, then there is no overhead in terms of migration cost. In fact, a job that is started on one processor, then pre-empted and resumed on another must have its context restored on the second processor. This can result in additional communication loads and cache misses that would not occur in the partitioned scheme where migration is not allowed.

3. Partitioned approaches use a separate run-queue per processor, rather than a single global queue. For large systems, the overheads of manipulating a single global queue can become excessive.

Figure 2.5 shows the architecture of a system in which it is implemented the partitioned scheduling.

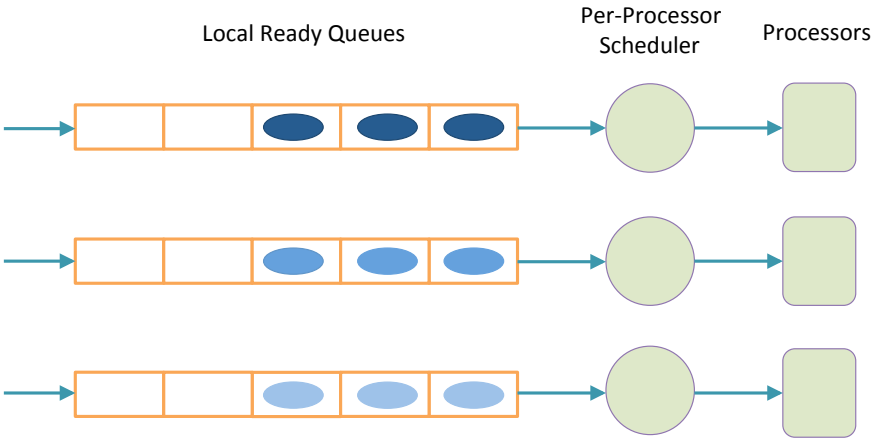


Figure 2.5: Multiprocessor system with partitioned scheduling [4]

Furthermore, by using a partitioned approach to multiprocessor scheduling, once an allocation of tasks to processors has been achieved, a wealth of real-time scheduling techniques and analyses for uniprocessor systems can be applied. On the other hand, the main disadvantage of the partitioned approach to multiprocessor scheduling is that the task allocation problem is analogous to the bin packing problem and is known to be NP-Hard [51].

2.3.4 Global Scheduling

Global scheduling has the following advantages compared to partitioned scheduling:

1. There are fewer context switches and pre-emptions when global scheduling

is used, this is because the scheduler will only pre-empt a task when there are no processors idle [52].

2. Spare capacity created when a job executing for less than its WCET can be utilised by all other jobs, not just those on the same processor.

3. Global scheduling is more appropriate for open systems, as there is no need to run load balancing or task allocation algorithms when the set of tasks changes.

Figure 2.6 shows the architecture of a system with a global scheduler that dispatches jobs to the appropriate core.

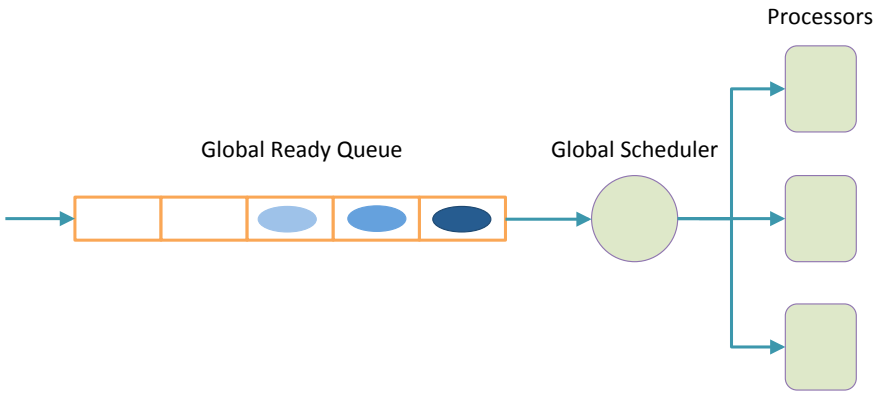


Figure 2.6: Multiprocessor system with global scheduling [4]

In the whole, depending on the hardware architecture, the overheads incurred by global scheduling can potentially be very high. The fact that jobs can migrate from one processor to another can result in additional communication loads and cache misses, leading to increased worst-case execution times that would not occur in the fully partitioned case. However, fully partitioned approaches suffer from the drawback that the available processing capacity can become fragmented, such that although in total a large amount of capacity is unused, no single processor has sufficient capacity remaining to schedule further tasks. In fact, with regard to partitioned scheduling, the maximum utilisation bounds is just 50% of the total processing capacity.

2.3.5 Hybrid Scheduling Approaches

The hybrid approaches combine elements of both partitioned and global scheduling and are respectively the clustered and semi-partitioned strategies.

The clustered approach first partitions the cores within multiprocessor platform into clusters and then each task is mapped on to a single cluster. Migration of a task's jobs is only allowed within the cluster to which the task is mapped. Clustering can be thought of as a form of partitioning with the clusters effectively forming a smaller number of faster processors to which tasks are allocated. In such case, capacity fragmentation represents less an issue if compared with partitioned approaches while the small number of processors in each cluster reduces global queue length and has the potential to reduce migration overheads. For example, processors in a cluster may share the same cache, reducing the penalty in terms of increased worst-case execution time of allowing tasks to migrate from one processor to another.

The semi-partitioned scheduling algorithms place various forms of restriction upon migration without forbidding it outright. Such algorithms are commonly called semi-partitioned or limited migrative scheduling algorithms. They may, for example, specify that no individual task is allowed to migrate between more than two processors or they may restrict the total number of migratory tasks in a system.

2.4 Chapter Summary

This chapter surveys various aspects of the standard real-time scheduling both in uniprocessor and multiprocessor platforms.

Section 2.1 contains a general introduction about real-time systems which consists of a discussion on the main concepts to devise real-time scheduling software. Section 2.2 reviews the main scheduling approaches on uniprocessor platforms. More precisely, it describes static and dynamic scheduling but also complementary techniques to increase the amount of jobs successfully processed, completed after their deadlines or to optimise some specific performance criterion. Lastly, Section 2.3 contains a review about the most used scheduling approaches on multiprocessor architectures.

Chapter 3

Related Work

This chapter review the main related work produced on mixed-criticality scheduling that is related to my research.

First of all, the chapter introduces the mixed-criticality task model with its successive extensions. Then, it presents the main fixed and dynamic priority scheduling both on uniprocessor and multiprocessor platfotms. Lastly, it also describes mixed-criticality scheduling with soft real-time tasks and that based on utility optimisation.

3.1 Mixed-Criticality Scheduling

Research on mixed-criticality scheduling was started by Vestal in 2007 to manage the challenge of accomplishing an effective and resource efficient a priori verification of safety-critical real-time systems with an increasingly integration of multiple functionalities [11, 12].

The mixed-criticality scheduling is built upon a novel approach to the a-priori estimation of tasks' properties in which multiple models are constructed for a task system, each of which true to a different level of assurance. The successive verification of functionalities is made at the level of assurance appropriate for the specific level. The initial task model consists of a set of periodic tasks that perform functions having different criticalities and requiring different levels of assurance [11]. Each task may have a set of different *Worst Case Execution Time* (WCET) estimates, each assured to a different level of confidence. The more confidence one needs in a task execution time bound, the larger and more conservative that bound tends to become in practice. The final aim was that to guarantee that safety-critical task instances do not miss their deadlines. The mixed-criticality approach allows the system developer to avoid

the usage of excessively conservative models to verify less critical functionalities and thus reduces the over-approximated estimates. Applying such modular certification enables to design systems that are verified correct, that make a more efficient usage of platform resources during runtime and that would have otherwise been deemed unschedulable using conventional analysis techniques. Guan et al. showed that such approach can be beneficial also in reducing cost and energy consumption [53].

The mixed-criticality task and system model permits to deal with multiple criticality levels. However, for simplicity, most of research work studies dual-criticality systems, i.e., systems with tasks having just two criticality levels, indicated in this dissertation as LO (low criticality) and HI (high criticality) with HI being more critical than LO. Since the HI task instances execution has to be guaranteed at a higher level of assurance than LO jobs, a common approach is to assume for LO tasks only the knowledge of easy to derive optimistic WCET estimates while for HI jobs also a higher level of assurance based on safe upper WCET bounds is assumed.

Therefore, the research regarding the mixed-criticality approach was initially developed to address fundamental questions regarding a priori verification. However, the Vestal model presents new challenges with regard to scheduling and fault tolerance. In fact, well designed mixed-criticality scheduling methods should seek to satisfy two, sometime contradictory, ultimate goals [54]:

1. Guaranteeing the execution of higher criticality jobs at the required level of assurance, even under very conservative assumptions.

2. Achieve high resource utilization during run-time.

Successive work has been done on extending the applicability of such approach to address also issues about run-time robustness.

The mixed-criticality scheduling protocols aim to overcome the limitations of standard priority based scheduling algorithms [55]. The active research challenge is to find ways to effectively combine the resource prioritisation based on criticalities with the scheduling priorities based on real-time constraints. The basic idea of mixed-criticality protocols is that as long as enough resources are available, the scheduling priorities are defined by a real-time scheduling protocol. In case of a resource shortage, e.g., a highly critical job overrunning its estimated WCET [15], the tasks' properties referred to as criticalities are used as the primary criterion to allocate resources.

3.2 Extensions and Applications of Mixed-Criticality Model

After the Vestal seminal work, research on mixed-criticality approach has evolved to generalise the initial system model in order to make the solutions proposed suitable for industry.

On one side, extensions to the classical mixed-criticality task model have been proposed. Baruah, Burns and Davis have presented generalizations to the standard 3-parameter sporadic task model in which multiple estimates are provided for one or even for each of the three task parameters [56, 57, 58]. As an example, in many cases, the minimum inter-arrival time of tasks is not known precisely a priori but it must be estimated and hence it makes sense in a mixed-criticality settings to specify multiple periods for tasks in which smaller values represent safer and more conservative estimates. Baruah studied this issue for the first time upon preemptive uniprocessor platforms by providing a task model for dual-criticality systems [54]. He devised a system model in which each task might have more than one estimate for its period but only one estimate for its WCET. Schedulability analysis techniques were provided afterwards first for fixed-priority [59] and then for EDF-based dynamic priority systems [60]. Burns and Davis also considered the scheduling of streams of real-time traffic characterized by multiple period parameters at different levels of pessimism within *Controller Area Network* (CAN) [61]. Their work investigated what form of mixed-criticality support should be integrated within CAN technology. Lastly, they defined mixed-criticality protocols that could form the basis of a *Trusted Network Component* (TNC) for CAN and derived a sufficient response-time analysis for such protocols and an optimal priority assignment scheme. Other system models consist of tasks that might have multiple relative deadlines with smaller (or sometime larger) relative deadlines for increasing criticality levels [62, 57].

It is also possible to apply similar generalisations to system resources, even different from processor capacity; for instance, it could be possible to make a priori predictions about the run-time energy usage of a system by modeling the likely energy consumption of components differently at different levels of assurance and using the appropriate models to validate functionalities of different criticalities. Other resources such as communication bandwidth or memory usage could be similarly considered.

Alan Burns and Robert I. Davis provided a historical overview and a general introduction about the mixed-criticality scheduling that also contains the gener-

alisations to the initial Vestal model and the most recent applications [56]. The key question raising from their work is how to reconcile the conflicting requirements of partitioning for safety and sharing for efficient resource usage.

Crespo et al. studied the possibility to use virtualisation as basis for building mixed-criticality partitioned software architectures [63]. Their work reviewed the challenges connected to systems with virtual partitions having different criticality that are executed in an independent way. Such systems are based on a hypervisor that provides temporal, spatial and fault isolation among partitions that contain components that have to be guaranteed at different assurance levels and on hierarchical scheduling as strategy to process jobs.

Cros et al. have studied how to manage criticality level information in ethernet networked systems [64]. In fact, a modern networked system can be thought as a system for interconnected applications of different criticalities. Nowadays, real-time industrial networks providing timing guarantees for applications of different criticalities often are built in separate physical infrastructures, one for each type of network at the price of cost, weight and energy consumption. The authors presented a criticality-change protocol in a clock synchronized switched ethernet network, in the case of two criticality levels. The main goal of the criticality management in such networked systems consists in providing *Quality of Service* (QoS) guarantees in terms of worst case end-to-end transmission delays, particularly for high critical messages.

3.3 Interpretations of Criticality

Despite the fact that research on mixed-criticality has already produced a considerable amount of work, the ultimate and definitive meaning of criticality is still subject of discussion within the scheduling community.

In fact, a task's criticality can be derived from different aspects. One possibility is to express the relative importance or relative utility of different services in a system as their criticality [6]. Another possibility is to express the relative level of assurance, for example, dictated by different development standards for safety critical or relevant systems, like DO-178C [7] in the avionics domain, ISO26262 [8] in the automotive domain, or IEC 61508 [9] in the automation domain as different levels of criticality. Lastly, Buttazzo defines the criticality as importance based on consequence of missing a deadline, that could be hard, firm or soft [10].

Ernst and Di Natale provided an explanation about the meaning of criticality

and a review about the mixed-criticality model in current real-time research [65]. They highlight how functional safety standards usually provide the basis to design industrial mixed-criticality systems. Each safety or design assurance level involves a certain likelihood to perform successfully the required functions under certain conditions and within a stated period of time. The definition of criticality levels is often obtained as a result of a *Failure Modes, Effect and Criticality Analysis* (FMECA) process. However, these standards focus on the safety targets while engineers normally focus on metrics such as *Size, Weight and Power* (SWaP) specifications. Such contrast grows with the autonomous driving and with the integration challenges derived from cyber-physical systems and *Internet of Things* (IoT).

3.4 Criticisms to the Mixed-Criticality Scheduling

The mixed-criticality scheduling is encountering growing attention and interest. However, after a great initial excitement from practitioners there has been also some apprehension regarding the real practical relevance of the mixed-criticality scheduling [66, 67].

First of all, such concerns pointed out how the mixed-criticality scheduling methods do not reflect the current practice with regard to ensuring robustness in safety-critical systems. From this point of view, the safety standards require that functions at any criticality level should not be affected by timing errors in other criticality levels. Conversely, the mixed-criticality protocols on uniprocessor architectures always guarantee the correct completion of higher criticality tasks at the expense of lower criticality ones. Moreover, most research efforts within mixed-criticality community are currently dedicated to the usage of scheduling methods for ensuring graceful degradation after timing faults while the main concern derived from safety standards remains the sufficient independence of components while optimizing design efficiency. Ernst and Di Natale also notice that the idea of representing a system using multiple models to validate different parts of it appears to currently be antithetical to the authorities and entities responsible for certification. In other words, it is not clear the reason for certification authorities to accept two WCET values for a task and two different processes for measuring them. However, Baruah noticed that, even if the most significant shortcoming is the incompatibility of the current mixed-criticality scheduling methods with current certification standards in safety-critical sys-

tems, this seems to be especially a social and cultural problem, rather than a technical one. In fact, [66] and [67] pointed out various parts of current standards that do not permit to represent a system as consisting of parts that have to be validated at different levels of assurance. However, they do not indicate any technical reasons that could prevent to incorporate such modeling approaches within the standards in the future.

A further criticism to some work produced by researchers in mixed-criticality scheduling is about the ambiguity of the criticality meaning. The word criticality is sometime used to represent several related but different concepts such as importance, confidence, safety integrity levels or even consequence related to a deadline missed. However, the term criticality has a specific technical meaning in the safety and certification community which does not always correspond to importance nor to the severity of the consequences of a failure in a function but that is the result of a more complex assessment and design process [67]. An analysis of the current mixed-criticality scheduling models from the safety assurance perspective is provided in [68]. It highlights how the key assumption behind most of mixed-criticality task models is the WCET confidence monotonicity, i.e., the higher the level of guaranteed assurance and the larger the relative WCET estimate shoud be.

3.5 Standard Real-Time Scheduling and Criticality Inversion Problem

The classical real-time scheduling community usually represent a task τ_i by using three-parameters as follows [26]:

$$\tau_i = \langle P_i, D_i, C_i \rangle$$

in which P_i represents the period, D_i the deadline and finally the C_i is the WCET. In such model, the scheduling decisions are made not considering the criticality of jobs but only the requirements indicated by a priority to determine what job has to be executed next. The priorities are usually assigned with the purpose of maximizing the schedulable utilization in order to respect the deadlines of all jobs within a job set. The utilization maximization approach of traditional real-time scheduling typically makes two important assumptions [13]:

1. All tasks are equally important and consequently also all jobs are equally important;

2. The utilization never goes beyond the allowable thresholds.

A remarkable way that scheduling analysis has been extended in the recent years considers the removal of assumption that all tasks and jobs that have to be scheduled have the same level of criticality [57, 13]. Moreover, the second assumption does not hold as there is the possibility that some jobs go beyond their specified C_i. The presence of criticality as task property has led to devise novel and more appropriate models in which one or more task paramaters can change according to the required level of assurance. Most of time, such models deal with systems having tasks that could be of high criticality or of low criticality, indicated in this dissertation as HI or LO tasks. The HI tasks have two WCET estimates, one optimistic and the other more conservative, indicated respectively by C_{LO} and C_{HI}. Within mixed-criticality systems, if there is ever a situation where it is possible satisfying the deadline of only one job, it is always expected to meet first the one of the higher criticality job. When high criticality jobs go beyond their optimistic allowable threshold, then it is possible to have incoming non critical jobs that have higher scheduling priority. This usually leads to transient faults and to a phenomenon named *criticality inversion* [13] in which high priority non critical jobs preempt low priority jobs that have a higher impact on the overall system correctness, e.g., with regard to safety, potentially leading the latter to miss their deadlines.

A very simple way to eliminate the criticality inversion is to simply assign priorities to tasks first according to their criticalities and then according to their timing requirements. This strategy is named *Criticality Monotonic* (CM) or *Criticality As Priority Assignment* (CAPA) [13]. This approach eliminates the criticality inversion in case a HI job overruns its C_{LO} but it can lead to a very poor utilization if the resulting task order turns out to be contrary to the best priority assignment to maximize utilization and it can generate priority inversions.

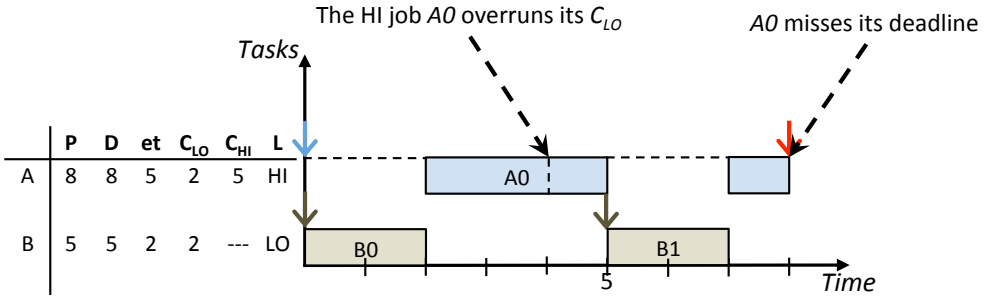


Figure 3.1: Criticality inversion example: job A_0 overruns its optimistic WCET, it is preempted by the higher priority LO job B_1 and misses its deadline.

Figure 3.1 shows an example of a mixed-criticality task set with implicit

deadlines which is scheduled with a standard real-time fixed-priority scheduler. Tasks A has criticality HI while task B has criticality LO. Priorities are assigned to tasks according to deadlines and jobs are processed using the the *Deadline Monotonic* (DM) algorithm. No jobs would miss their deadlines as long as they complete within their C_{LO} estimates. However, the lack of a runtime protection mechanism to ensure the completion of highly critical instances that exceed their optimistic time thresholds leads to a deadline missed. It is possibile to notice that the HI job A_0 exceeds its optimistic WCET at time $t = 4$ and is preempted by the higher priority LO job B_1 at time $t = 5$. Because of this, A_0 does not complete within its deadline.

The final aim of mixed-criticality scheduling protocols is mainly that to protect the HI jobs execution from the interference of higher priority LO jobs due to resource shortages. The current status of the art is that to prevent LO instances from preempting jobs with higher impact on the system correctness by abandoning the first ones whenever the timeliness of latter in meeting their deadline is endangered. This approach guarantees that high criticality jobs meet their deadlines and complete their execution. However, LO tasks are still relevant and simply abandoning their instances means decreasing too much their service level. Therefore, deciding how to solve the criticality inversion problem basically means deciding how to degrade lower criticality tasks in favour of the most critical ones.

3.6 Fixed-Priority Mixed-Criticality Scheduling

This section reviews the work that has been produced on the field of mixed-criticality scheduling algorithms that use fixed-priority assignment strategies on uniprocessor platforms.

In 2011, Baruah et al. extended the Vestal's model by proposing the *Adaptive Mixed Criticality* (AMC) together with response-time analysis techniques for constrained deadline dual-criticality task sets [69]. Such techniques have been recently extended to manage also sets of tasks with arbitrary deadlines [70]. The AMC scheduling algorithm requires a monitor to check how long each individual job executes. The scheduling protocol works in two execution modes, high criticality and low criticality that are indicated respectively with HI and LO, as described below:

1. There is a criticality level indicator Γ, initialized to LO.

2. While $\Gamma = LO$, at each instant the waiting job generated by the task with highest priority is selected for execution.

3. If the currently executing job does not complete within its optimistic WCET estimate, then the system changes execution mode and $\Gamma = HI$.

4. Once $\Gamma = HI$, all low criticality jobs will not be executed. Henceforth, at each instant the waiting job generated by the highest priority HI task is selected for execution.

5. An additional rule can specify the circumstances when Γ gets reset to LO, e.g., if no HI jobs are active at some instant in time.

In 2013, Li et al also extended the response time bound techniques and the AMC protocol to work with multiple criticality levels [71]. In 2017, Guo et al. studied the sustainability of various mixed-criticality scheduling tests both in uniprocessor and in multiprocessor systems and found that AMC-rtb is sustainable with regard to all parameters, including the criticality level [72]. The AMC protocol assumes that once the system goes into the HI mode, then all LO task instances will be abandoned and the system will remain in that mode. However, the sudden discard of LO jobs during the HI mode can cause serious service interruptions and significant performance loss, especially for control systems where the performance of controllers is mainly affected by the execution frequency and period of control tasks [73]. To reduce the impact on lower criticality tasks, the AMC protocol switches back to the initial execution mode as soon as an idle instant occurs and all LO jobs can be processed again with their timely execution [69].

Going back to the LO starting mode only in case of idle instants leads to a high amount of jobs interrupted or abandoned and this is still not satisfactory. Different complementary ways of guaranteeing a higher level of service for LO tasks have been proposed, e.g., extending their periods and/or deadlines like in the elastic task model [74] or reducing their execution times by switching to a simpler version of the software [75].

The *Priority May Change* (PMC) strategy has been proposed to better manage the situations in which higher priority LO tasks could preempt lower priority HI tasks in case of transient faults [76]. The AMC algorithm assigns a single priority to each task by considering together both LO and HI criticality modes whereas PMC computes priorities in two steps. Firstly, priorities are assigned to the tasks according to some predefined policy like deadline monotonic as they would in a regular task system [32]. These priorities are used by the runtime

dispatcher while the system is in LO execution mode. Once the system switches to HI mode, HI task priorities are re-assigned according to a priority ordering policy that is optimal for tasks with release jitter [77].

In 2014 Fleming and Baruah proposed a scheme in which the system designers can assign to lower critical functionalities a utility that is used to decide in which order their instances have to be suspended during the HI mode [78]. Such method allows to the system designer to control how non-critical functionalities degrade after the critical ones overrun their optimistic time threshold. The utility value is assigned as an ordinal scale [79] to provide a predefined order in which LO task instances will be dropped, least important task instances will be dropped first. The authors adapt the Audsley's priority assignment technique [80] to assign lower priority to lower utility LO tasks. Such protocol allows to increase performances for LO tasks and to process them for a significantly increased amount of time.

Somehow, the former methods considered so far allow for LO task invocations to execute after a criticality mode change but they are mainly best effort and do not have a predefined minimum threshold guaranteed for lower critical tasks. Since most hard real-time systems could miss some deadlines provided that it happens in a known and predictable way, the *Adaptive Mixed Criticality with Weakly-Hard constraints* (AMC-WH) was introduced in 2015 [81] and represents an extension of AMC [69] that integrates the notion of weakly-hard constraints. The definition of weakly-hard real-time system was given in 2001 [82] to indicate systems in which hard real-time tasks are permitted to miss some deadlines as long as the number of missed deadlines is strictly bounded. This work was based also on research on soft real-time systems [83, 84]. The AMC-WH is a scheduling policy that allows a number of consecutive instances per LO task to be skipped during the HI execution mode. This reduces the overall system load, frees more resources for safety-critical tasks and provides a degraded but guaranteed minimum quality of service for LO tasks upon a criticality mode change. The number of skips permitted and the number of subsequent deadlines that must be met can be a requirement deduced either from the design of a control algorithm[85] or from physical properties of the system. Empirical evaluations demonstrated that AMC-WH outperforms previous policies and accommodate the continued execution of LO tasks without compromising the assurance requirements for HI tasks.

However, as all former AMC-based methods, AMC-WH does not provide a fast recovery from the HI criticality mode since it is still necessary to wait for idle instants to go back to the starting LO execution mode.

Restoring the normal system execution mode avoids the abandonment of

new incoming LO jobs but the amount of instances aborted in case of resource shortage could be still very high since there is no control on when the idle instant will be. The *Bailout Protocol* (BP) improves AMC with the introduction of a fast and effective control mechanism to speed up the entering of the initial LO execution mode, called *Normal* mode [16, 17]. BP still has a HI execution represented by the Bailout and Recovery modes. The protocol aims to restore the Normal mode as soon as possible to minimize the number of LO jobs that miss their deadlines or are not executed at all. The Normal mode is restored not only at the occurrence of an idle instant but also according to the value of a *Bailout Fund* (BF) variable. The actual number of lower critical instances that will not be started depends on the size of BF, on number and execution time of LO instances and eventually on time needed for recovery. Once the system is back to the starting mode, all less critical functionalities start again to be processed with their full timely behavior. The strength of this protocol is that to speed up the entering of LO criticality mode, where all jobs can start and being processed. As a result, BP enlarges the duration of the Normal execution mode where all jobs can be processed with their full timely behaviour but still abandons LO jobs released during the execution of high criticality execution modes. Furthermore, BP allows to LO jobs released in Normal mode to continue to execute in Bailout and Recovery modes, even after their deadline as long as they do not exceed their C_{LO}.

The main weakness of BP is that to immediately drop low-critical instances after a HI job overruns its C_{LO}. Because of this, the percentage of LO jobs that miss their deadlines is still high. In the whole, abandoning lower criticality instances in case of resource shortage does not give the robustness required since some level for LO services should be maintained as they are still important for mission completion. There are two complementary strategies that help to reduce the number of times that a given system enters into Bailout mode, and the amount of time that it spends in such mode, hence reducing the number of LO instances that miss their deadlines or are abandoned. An approach to increase the amount of jobs scheduled was introduced by Santy et al [86]. Burns and Baruah have further refined it and adapted it to work with mixed-criticality protocols [75]. Such method exploits the system slack time by scaling up the C_{LO} of HI tasks without making the system unschedulable. This method effectively increases the execution time budgets while ensuring that the system remains provably schedulable according to AMC-rtb analysis [69]. If used together with the BP, the resulting protocol is named *Bailout Protocol - Slack* (BPS). More recently, Bate et al. have also integrated BP with a second complementary tech-

nique [17]. Such approach consists of an online update of the optimistic time budget that is made by exploiting the CPU spare capacity at runtime. These techniques allow to reduce both the number of times and the duration the system executes in HI modes. The most important property of any scheme for exploiting gain time is that the schedulability of HI tasks must not be affected. A number of mechanisms exist that can make this gain time available for use by other jobs without affecting the schedulability. The method used with BP operates *only* during the Normal mode and the gain time g_i of a LO job j_i is defined as follows:

$$g_i = C_{LO} \; - \; et(j_i) \tag{3.1}$$

Passing the gain time from one job to the successive makes less likely that jobs requiring more computing time than expected will actually exceed their C_{LO} budgets. On one side this increases the probability that LO jobs complete successfully instead of being dropped. On the other hand, it makes less likely that the system enters the Bailout mode because of the HI jobs overrruns. It is worth to note that in Bailout mode, the gain time mechanism is not used, since the BP effectively makes use of gain time to hasten recovery. The *Bailout Protocol with Gain time* (BPG) is derived from the integration of BP with the gain time collection at runtime. By combining simultaneously both complementary methods with BP, the authors have also introduced the *Bailout Protocol - Slack and Gain Time* (BPSG). The benefit of using such complementary techniques results in the increase of the overall service quality for lower critical tasks, provided by increasing the number of LO task instances correctly processed.

3.7 The AMC-rtb Analysis

The schedulability analysis used to create task set that have to be processed with AMC and BP consists of a three step verification process that guarantees the schedulability of each mixed-criticality constrained deadline task at the required level of assurance in each execution mode [69]. The analysis has the following three steps, each employs the appropriate test:

1. Verifying the schedulability of every task during the low-criticality execution mode:

$$R_i^{LO} = C_{LO} + \sum_{j \in \mathbf{hp(i)}} \left\lceil \frac{R_i^{LO}}{P_j} \right\rceil C_{LO,j} \tag{3.2}$$

2. Verifying the schedulability of HI tasks during the high-criticality execution mode:

$$R_i^{HI} = C_{HI} + \sum_{j \in \mathbf{hpH(i)}} \left\lceil \frac{R_i^{HI}}{P_j} \right\rceil C_{HI,j} \tag{3.3}$$

3. Verifying the schedulability of HI tasks during the criticality change itself:

$$R_i^{HI} = C_{HI} + \sum_{j \in \mathbf{hpH(i)}} \left\lceil \frac{R_i^{HI}}{P_j} \right\rceil C_{HI,j} + \sum_{k \in \mathbf{hpL(i)}} \left\lceil \frac{R_i^{LO}}{P_k} \right\rceil C_{LO,k} \tag{3.4}$$

where **hpH(i)** is the set of HI tasks with priority higher than that of task τ_i and **hpL(i)** is the set of LO tasks with priority higher than that of task τ_i. Furthermore **hp(i)** is the union of **hpH(i)** and **hpL(i)**.

As showed above, to verify the first two steps the standard response time analysis is applied while for the third assumption a new analysis named *Adaptive Mixed Criticality - Response Time Bound* (AMC-rtb) was introduced in [69]. The third phase checks the schedulability of the progress of criticality change. If an instance of a HI task τ_i exceeds its optimistic WCET, a change to the Bailout mode must occur at or before R_i^{LO} which delimits the maximum interference from higher priority LO jobs. If, for any HI task, $R_i^{HI} \leq D_i$ during the switch to the high criticality execution mode, then the task will remain schedulable once such execution mode is fully established.

3.8 Dynamic-priority mixed-criticality systems

The mixed-criticality protocols in which priorities are assigned dynamically are based on adaptations of the standard EDF to schedule tasks having multiple timing parameters per criticality level.

The *Earliest Deadline First with Virtual Deadlines* (EDF-VD) was introduced as a generalisation of standard EDF to manage sets of mixed-criticality sporadic tasks with implicit deadline upon preemptive uniprocessor platforms [87, 88]. A sufficient test was also provided to check whether a task set is feasible with EDF-VD. If a task set is deemed schedulable, then an additional period parameter $\hat{P}_i$ is computed for each HI task, with $\hat{P}_i \leq P_i$. The EDF-VD works in two criticality execution modes, LO and HI, indicated by the value of a variable Γ.

The scheduling protocol works as specified below:

1. There is a criticality level indicator Γ, initialised to LO.

2. While $\Gamma = LO$:

   - At each instant, the waiting job with the earliest absolute deadline is selected for excution. Deadlines are computed considering the period P_i for LO jobs or the modified period $\hat{P}_i$ for HI jobs.
   - If the current-executing job executes for more than its optimistic WCET without signaling completion, then the system switches to HI criticality mode and $\Gamma = HI$.

3. Once $\Gamma = HI$:

   - HI jobs are scheduled considering their initial periods (and relative deadlines) P_i.
   - LO jobs are abandoned.

4. An additional rule could be specified to speed up the switch back to the starting LO mode, e.g., this could happen if no HI job is active at some instant in time.

In 2013, Su et al. introduced the *Elastic Mixed-Criticality* (E-MC) [89] to address the issue of abrupt service interruption experienced with mixed-criticality scheduling policies and to provide minimal service guarantees for low-criticality tasks. The main idea underlying such model is that to have tasks with variable periods as in the *Elastic Task Model* [90]. The major difference between the E-MC and other mixed-criticality task models is in the way of representing low-criticality tasks. Within E-MC, low-criticality tasks have a couple of periods that are associated respectively with the desired period and the minimum service level required for that task. Specifically, the largest period of a low-criticality task represents its minimum service requirement. The same work also introduced a novel EDF variant to handle mixed-criticality tasks, the *Early-Release Earliest Deadline First* (ER-EDF), which allows to release early and more frequentently the LO tasks to improve their service level. With ER-EDF, low criticality tasks can be released earlier than their largest periods without sacrificing the timeliness of high criticality tasks. Compared with the EDF-VD scheduling algorithm, the simulation results show that the ER-EDF can successfully schedule much more task sets.

3.9 Mixed-Criticality Systems on Multiprocessor Architectures

The current trend towards the integration of cores into multi-core architectures allows to tasks having different criticalities to run on the same platform. This raises new challenges due both to potential task interference among mixed-criticality tasks and to verification and certification of platform subsystems. On multi and many-core platforms that run applications of different mixed-criticality, all applications have to be certified to the highest level of criticality, unless they are sufficiently isolated. Isolation enables individual certification of applications and cost-efficient re-certification of single applications after an update.

However, a holistic architecture for the seamless mixed-criticality integration encompassing distributed systems, multi-core chips, operating systems and hypervisors is still an open research problem. Obermaisser et al. have described the state-of-the-art of mixed-criticality systems, ranging from distributed to multi-core chips, and discussed the ongoing research within the European project DREAMS on a hierarchical mixed-criticality platform with support for strict segregation of subsystems, heterogeneity and adaptability [91].

The *Integrated Dependable Architecture for Many Cores* (IDAMC) platform [18, 19] was introduced to run multiple mixed-critical applications on a single multi-core platform. IDAMC is a *Network on Chip* (NoC) tiled architecture that provides spatial and temporal isolation. It supports safe sharing of resources, a transparent mapping of applications to available resources and isolation of mixed-criticality applications on a shared platform. The IDAMC monitoring mechanism allows to isolate a possible faulty low critical application/tile to guarantee the timing of high critical applications running on other tiles. IDAMC allows to isolate highly critical tasks against faulty low critical tasks and, by guaranteeing the timing of mixed-critical applications, to reduce their cost for certification and re-certification.

Su et al. analysed the performances of the *Elastic Mixed-Criticality* (E-MC) approach on multicore systems with identical cores that can share different levels of on/off-chip caches [20]. E-MC was first introduced together with the *Early-Release EDF* (ER-EDF) on uniprocessor systems to improve the service level provided for low-criticality tasks. The authors first investigated the schedulability of E-MC tasks under various well-known task-to-core mapping heuristics and then compared ER-EDF with the Global EDF-VD scheduler. Results show

that the proposed E-MC with the early-release scheme can significantly improve the service levels of low-criticality tasks while Global EDF-VD may severely and negatively affect them by canceling most of their task instances at runtime, especially for systems with more cores.

Legout et al. proposed the LPDPM-MC approach to reduce the energy consumption of multiprocessor mixed-criticality embedded systems by continuining to guaranteeing that high-criticality tasks meet their deadlines [21]. LPDPM-MC reduces energy consumption by increasing the amount of low-criticality jobs that miss their deadlines. Since task instances may not use entirely their WCET estimates and low-criticality tasks are assured at a lower level, such approach uses part of the time budget of low-criticality tasks to find an appropriate trade-off between the number of missed deadlines of low-criticality jobs and energy consumption. The LPDPM-MC reserves offline only a percentage of the entire WCET of low-criticality tasks. Then, the spare time generated can be used online by other low-criticality tasks to meet their deadlines. The approach uses the LPDPM to minimize the static energy consumption via linear programming [92]. The percentage of deadline misses must be chosen by the system designer as an input parameter and depends on the criticality level of tasks, i.e., the lower the criticality level is, the higher this percentage can be. The designer can control the aggressiveness of the solution depending on whether the focus should be on reducing the energy consumption or on low-criticality deadline misses.

Thekkilakattil et al. proposed a fault tolerant approach to mixed-criticality real-time scheduling that considers the recommendations given by the reliability studies, e.g., hardware reliability studies like *Functional Hazard Analysis* (FHA) and *Zonal Hazard Analysis* (ZHA), to improve the overall system reliability and safety [93]. FHA and ZHA are usually used for safety critical systems to ensure that the proposed redundancies on the hardware components, e.g., wires and communication sub-systems, indeed exist. The authors consider a distributed real-time architecture with identical multi-processors that communicate over a reliable communication media and that are synchronized via software. Such approach aims to provide real-time guarantees for the critical tasks offline and to ensure flexibility for the non-critical tasks.

Yun et al. proposed a software-based memory throttling mechanism to bound the task interference in multicore mixed-criticality systems with shared memory [94]. Such work is of particular interest because the existing research on multicore mixed-criticality scheduling ignores the effects of resource sharing on the response times of applications. If tasks on different cores access simultaneously, e.g., to a memory bus, timing interference among applications of different

criticalities cannot be avoided. Yun proposed a protocol in which the cores where low-criticality tasks are executed are assigned a limited memory budget so that schedulability of the high-criticality tasks is guaranteed and meanwhile the impact on the low-criticality tasks is kept under control. Furthermore, Giannopoulou et al. introduced a scheduling policy for resource-sharing multicores that prevents timing interference among applications of different criticality levels [95]. This is achieved by allowing only a statically known set of applications of the same criticality to be executed across the cores at any time. This enables timing isolation despite resource sharing without any need for hardware support.

A successive work discussed how to combine the *Flexible Time-Triggered and Synchronisation* (FTTS) mixed-criticality scheduling policy with an optimization method to partition tasks to cores and to statically map memory blocks, i.e., task data and communication buffers, to the banks of a shared memory architecture [96]. The authors presented a heuristic approach to partition and schedule mixed-criticality task sets on a multi-core architecure together with a annealing-based algorithm to statically map the task data and the communication buffers to global memory banks, such that the effect of bank sharing on the task response times is minimized. Lastly, since the above two optimization problems are interdependent, they propose two possible approaches for integrated design optimization.

Burns et al. adapted the traditional cyclic executive scheduling on multi-core systems to handle tasks having up to five criticalities [97]. The authors considered both partitioned and global scheduling schemes and criticality monotonic as priority assignment. They used a strategy in which, at any given instant in time, all the processors are only allowed to execute code of the same criticality level as this rules out the possibility that less critical code interferes with the execution of more critical code in accessing shared resources [95]. For partitioned scheduling, no optimal polynomial-time algorithms can be devised since the problem is NP-hard in the strong sense [98]. Therefore the authors have investigated the usage of common heuristics like best-fit and worst-fit to effectively map application tasks to the multi-core cyclic executives. Conversely, for global scheduling, it was proposed a sufficient schedulability test that determines whether a given mixed-criticality system is feasible together with an algorithm that actually constructs a schedule in case of schedulability. Lastly, they also estimated the reduction in schedulability that arises from the requirement that only code of the same criticality executes at the same time.

Izosimov and Levholt presented a new metric to design and assess mixed-criticality multi-core systems without changing the development flow and prac-

tice [99]. The primary goal in development of such metric was to provide a tool for engineers and safety managers in taking decisions with respect to the mixed-criticality and help to justify and judge a particular solution for safety-critical system design. Safety standards usually focus on reduction in severity of faults. However, due to the increasing systems integration and to the presence of multi-core platforms, complexity and performance should also be considered in taking design decisions about modern systems. Thus, the proposed mixed-criticality metric balances the reduction in severity of faults together with the implications on reduction in performance and increase in system complexity.

3.10 Mixed-Criticality Systems with Soft Deadlines

In many real-time applications, the consequences of missing a deadline vary in severity from task to task. As an example, in RTCA DO-178B a system safety analysis assigns to each task a criticality level (ranging from A to E) [100] and erroneous behavior of a level A task might cause loss of a critical function required to fly and land safely with possible catastrophic consequences while an erroneous behavior by a level E task might at worst cause inconvenients with no impact on safety nor on aircraft operations.

Within such systems, a deadline miss of tasks at different criticality levels leads to different consequences for the system functioning. Generally, the higher the task criticality level, the higher the impact on the system safety or correctness caused by the deadline missed. Conversely, for lower criticality levels it can be possible to have some deadlines missed with no impact on system safety. Because of this, it can be also possible to allow to a predefined amount of LO jobs to complete after their deadlines as long as such exceeding time is reasonably bounded.

The mixed-criticality scheduling methods on uniprocessor platforms use different task parameters estimates at different level of assurance with higher criticality tasks guaranteed under more conservative assumptions. Such methods ensure the correct completion of HI jobs in all operating conditions and always at the expense of lower criticality instances. As a result, the mixed-criticality scheduling protocols on uniprocessor platforms are emerging as protocols designed to handle all tasks as hard real-time tasks but that, in case of resource shortages, lead to a degradation of service provided to lower criticality tasks. This approach guarantees that HI jobs never miss their deadlines. However,

ensuring the correct system functioning by guaranteeing the correct completion for all highly critical jobs by just abandoning lower criticality instances does not allow to control the amount of abandoned instances [23].

Part of the research work that has been produced to address the sudden degradation of LO tasks adopts strategies used for soft real-time systems to reduce the impact on less critical tasks by maximising the amount of LO jobs scheduled.

A strategy to avoid the abrupt degradation of lower criticality services is to adjust at runtime the arrival rates of LO tasks to provide a reduced but still acceptable level of service. With this regard, as stated in Section 3.8, Su et al. introduced the *Elastic Mixed-Criticality* (E-MC) task model together with the *Early-Release EDF* (ER-EDF) scheduling algorithm [101]. The E-MC model treats HI tasks as hard tasks with not modifiable and strict timing parameters that have to be scheduled on time and always meet their deadlines while LO tasks are treated as tasks with flexible arrival rates. The key idea underlying such model is that to have variable periods for LO tasks where the minimum service requirements are represented by their largest periods.

Schneider et al. proposed a multi-layered scheduling scheme for cyber-physical systems that consist of a mix of hard and soft real-time tasks [102, 103]. Time-critical applications have hard deadlines that have to be always guaranteed while applications not strictly related to deadlines are rather scheduled considering the *Quality of Control* (QoC). Traditional scheduling policies such as deadline monotonic can guarantee timing deadline constraints but do not allow for QoC optimized schedules. The authors presented a scheduling algorithm that ensures both that all hard real-time constraints are met and that the overall QoC for the remaning applications is maximized. The multi-layered scheduling scheme introduced implement the appropriate scheduling strategies for each type of criticality present in the system, i.e. mixed-criticality task sets are scheduled at different layers. In particular, time-critical real-time tasks are scheduled according to the worst-case assumptions to complete within their deadlines while the remaining tasks are scheduled with respect to optimized QoC. Results show that, compared with the standard deadline monotonic scheduling, this approach significantly improves the overall QoC while guaranteeing schedulability for tasks with hard deadlines.

Moreover, Mollison et al. developed an architecture to schedule periodic mixed-criticality real-time tasks on multiprocessor platforms in which tasks at each criticality level are scheduled according to different scheduling policies [104]. The cyclic executives and partitioned EDF schedulers are used to schedule tasks

at the highest criticality levels which perform safety-critical operations. Global EDF containers are used for tasks at medium criticality levels in charge of mission critical operations. Lastly best effort scheduling is used for lowest criticality level tasks. Their architecture proposes a system in which the higher criticality tasks are hard real-time tasks and their jobs must never miss their deadlines while lower criticality tasks are soft real-time tasks and for such instances some deadline misses or even a controlled completion after the deadline are tolerable.

A mixed-criticality architecture in which tasks are scheduled according to their criticality was also introduced by Selicean [105]. They proposed a simulated annealing based algorithm for the optimization of time-partitions for mixed-criticality real-time distributed embedded systems. Applications are represented by tasks, each with its own criticality assigned according to *Safety-Integrity Level* (SIL) [9]. The algorithm considers that the applications are separated using a temporal and spatial-partitioning scheme. Safety-critical applications are scheduled using static-cycling scheduling while the non-critical applications are scheduled using fixed-priority preemptive scheduling. Although their work addressed hard real-time applications, the authors explicitly state that non-critical tasks, i.e., those corresponding to the lowest SIL levels, can also be represented by soft real-time tasks that are handled by the CBS server.

3.11 Utility-based Optimisation of Mixed-Criticality Systems

In chapter 2, I presented the utility functions approaches in the context of the standard real-time scheduling. However, the first usage of how to use utility functions to optimise predefined performance parameters within mixed-criticality systems is due to Kirner by means of the *Tolerance-based Real-Time Computing Model* (TRTCM) [1, 22]. His work outlines how to optimise the *Quality of Service* (QoS) by maximising the overall system utility in case of standard and mixed-criticality real-time services. Such model could also be used to optimise a number of criteria such as latency, throughput and jitter within systems in which tasks have different criticality requirements. Then, the TRTCM has been further analysed and developed for adaptation of mixed-criticality systems with periodic task sets on uniform multiprocessors [6].

The TRTCM differs from the traditional utility accrual via *Time/Utility Functions* (TUF) mainly because it specifies a tolerance range in which predefined service parameters could be degraded till the least point for which it

is still acceptable. The tolerance interval is necessary to smoothly degrade the service quality provided to each individual task in a task set. Each task in the system, regardless of its criticality, has got a tolerance interval. First of all, it is necessary to choose what performance parameter to optimise, then the *Quality of Service* (QoS) provided according to those parameters could be adjusted at runtime. To optimise the overall system utility in case of resource shortages, the TRTCM allows also for an acceptable and smooth degradation of HI tasks. Such feature is not supported by the majority of existing mixed-criticality approaches. In fact, currently, the scheduling of mixed-criticality task sets is built upon a static service guarantees at different certification levels and this leads to drastically reduce the amount of LO jobs scheduled whenever the correct completion of HI instances must be assured. On the other hand, TRTCM is based on a system utility optimisation in presence of faults that aims to find the right trade-off between higher and lower criticality tasks by exploiting their tolerance range. As a result, the TRTCM allows for reconfiguration of mixed-criticality systems in case of runtime failures. The resource shortage is a necessary condition to take advantage from TRTCM and for its applicability.

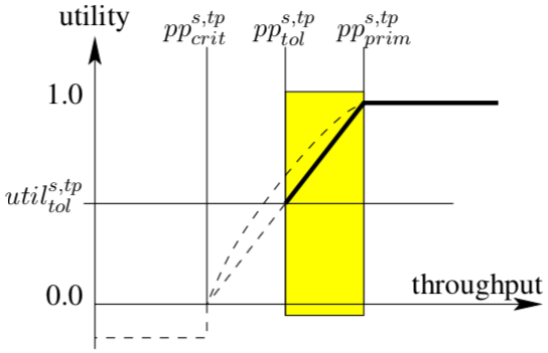


Figure 3.2: Example of TRTCM utility function for throughput

The utility functions of TRTCM are not limited to *latency* but it is possible to describe multiple functional properties such as *throughput*, *jitter* and *energy consumption*. As an example, Figure 3.2 shows an example of TRTCM utility function usage to model the throughput of each service within the system. The key idea is to use a simplified two-point function identified by the interval $[pp^{s,tp}_{tol}, \ldots, pp^{s,tp}_{prim}]$ that is called tolerance range (with $pp^{s,tp}_{crit} < pp^{s,tp}_{tol} < pp^{s,tp}_{prim}$). The aim is to maximise the system utility according to the constraints given by the application requirements and by the available resources. Equation 3.5

represents the objective function corresponding to the Figure 3.2:

$$SU_{tol}^{tp} = \sum_{s \in \text{SERVICES}} CRIT_s \cdot util^{s,tp}. \tag{3.5}$$

in which the coefficients $CRIT_s$ are positive real numbers and $util^{s,tp}$ are the throughput service utilities. The system and resource constraints are respectively showed below:

$$\sum_{s \in Services} et_s \cdot pp_{bound}^{s,tp} \leq \sum_{p_i \in Cores} C_i \tag{3.6}$$

$$pp_{bound}^{s,tp} \geq pp_{tol}^{s,tp} \tag{3.7}$$

$$util^{s,tp} \leq 1 \tag{3.8}$$

$$util^{s,tp} \leq pp_{bound}^{s,tp} \cdot k^{s,tp} + q^{s,tp} \tag{3.9}$$

The inequation 3.6 bounds the total workload by the total computing capacity of all processing elements $p_i \in Cores$. The remaining inequations represent the minimal acceptable throughput of $pp_{tol}^{s,tp}$ and the utility constraints.

3.12 Chapter Summary

In this chapter, I reviewed the related work of this dissertation. I provided an historical introduction which explains the issues about the a priori verification, the current challenge in providing run-time robustness and the motivations that led to devise different and novel task models.

Then, I explained the problem of criticality inversion arising when the standard real-time scheduling techniques are used to process set of tasks having different criticality. In such cases the standard priority driven scheduling algorithms, that give precendence to most urgent jobs even in cases of resource shortages, lead the most critical ones to miss their deadlines. I also considered the work produced in the field of fixed and dynamic priority systems both on single and multi-core architectues, including also mixed criticality systems with soft real-time tasks.

Lastly, I reviewed the recent TRTCM advances and its integration with the mixed-criticality systems to optimise system performance parameters such as latency, throughput and jitter.

Chapter 4

The LBP Protocol

This chapter introduces the *Lazy Bailout Protocol* (LBP), which is a mixed-criticality protocol to schedule set of tasks having two criticality. Compared with the former *Bailout Protocol* (BP), LBP increases the amount of low criticality jobs that are scheduled. In fact, low criticality jobs that are released during the high criticality execution modes or those that exceed their WCET estimates are inserted in a low-priority queue instead of being abandoned. Such jobs will be scheduled during the system idle time. This allows to rescue and to execute a considerable larger amount of jobs and to reduce the impact of resource shortage.

Section 4.1 presents the task and system model together with the assumptions under which the protocol works correctly. Section 4.2 shows the LBP architecture and explains what are its execution modes. It also contains a detailed description of each of its execution mode. Section 4.3 introduces an LBP variant, the *Soft Lazy Bailout Protocol* (SLBP), that treats lower criticality jobs as soft real-time jobs. SLBP can be used in cases for which a reasonable late completion for lower criticality jobs is better than no result at all. Section 4.4 explains how to integrate LBP and SLBP with scheduling techniques to increase the number of LO jobs scheduled. Finally, Section 4.5 concludes and summarises the chapter.

4.1 System model

In the following it is described the system model used for task sets. It is assumed a dual-criticality system, which consists of multiple tasks, where each task has a criticality $l \in \{LO, HI\}$ with HI being of higher criticality than LO. As discussed in Chapter 3, the criticality of a task can be derived by different means but no specific interpretation of criticality is assumed, as this is orthogonal to the scheduling method. Furthermore, it is assumed that the processor is the only

resource that is shared among tasks, and that the overheads due to the scheduling operations and context switches can be bounded by a constant included within each task worst-case execution times. The system schedules sets of independent and periodic tasks τ on uniprocessor platforms and each task set consists of two sub sets:

$$\tau = \tau_{LO} \cup \tau_{HI} \tag{4.1}$$

with

$$\tau_{LO} = \{\tau_i \in \tau \mid l_i = LO\} \tag{4.2}$$

$$\tau_{HI} = \{\tau_i \in \tau \mid l_i = HI\} \tag{4.3}$$

where τ_{HI} is the subset of tasks that are highly critical and τ_{LO} is the subset of tasks that are not highly critical within the system.

The tasks represent scheduling units that the system has to perform. An individual task $\tau_i \in \tau$ is represented by the following tuple:

$$\tau_i = \langle P, D, C_{LO}, C_{HI}, L \rangle$$

where P is the period, D is the relative deadline, C_{LO} and C_{HI} are respectively the optimistic and the pessimistic worst case execution time estimates and $L \in \{LO, HI\}$ refers to the criticality.

A job is an instance of a task at runtime, i.e., a job represents the actual object processed by the scheduler and inherits almost all properties from the task that generates it plus the arrival time A as below:

$$j_i = \langle A, D, C_{LO}, C_{HI}, L \rangle$$

The LO tasks and, as a consequence, their relative jobs do not have a known safe WCET bounds C_{HI}, since safe worst-case execution times are rather costly to obtain and thus provided only for HI tasks. Once it finishes its execution, each job j_i has got a computation time $et(j_i)$ that can vary for each specific job of the same task. The job set produced by an individual task τ_i is indicated by $J(\tau_i)$ while $J(\tau)$ is the job set produced by all tasks belonging to the task set τ. Therefore, τ represents the set of activities that have to be performed by the system while J represents the set of concrete process instances that have to be considered by the scheduler.

The jobs produced via the task set are scheduled according to the standard fixed-priority fully pre-emptive real-time scheduling. However, the traditional fixed-priority scheduling is unaware of criticality of task instances and scheduling decisions are only made according to priority that indicates the job timing requirements. Therefore, it is also used a protocol that considers the task's criticality to meet the mixed-criticality requirements. The following assumptions are made about the task set and the underlying real-time scheduler, i.e., fixed priority fully pre-emptive scheduling:

Assumption 1: all HI and LO jobs together are schedulable with the underlying real-time scheduling method with respect to their C_{LO}.

Assumption 2: all HI jobs alone are schedulable with the underlying real-time scheduling method with respect to their C_{HI}. Since C_{HI} is a safe WCET bound, i.e., $et \leq C_{HI}$, this assumption also implies that the HI jobs alone are schedulable with respect to their actual execution time.

Assumption 3: all HI jobs are schedulable with respect to their C_{HI}, while also assuming the execution of all LO jobs with respect to their C_{LO} having arrived before any HI job j_j overruns its $C_{LO,j}$.

Note that Assumption 3 is required so that while LO tasks are allowed to run within their C_{LO}, it is still ensured that all HI tasks are still schedulable within their C_{HI}. Assumption 3 is based on jobs rather than tasks as it covers the moment in time when a HI task overruns its C_{LO}. Also note that Assumption 2 is just a weaker case of Assumption 3, without the LO tasks considered.

4.2 The LBP Protocol

The standard BP is an adaptive protocol to schedule mixed-criticality job sets. The strength of BP is that to provide an effective and fast control mechanism to go back to the LO criticality mode, where all jobs can start and being processed. However, the main weakness of BP is that to immediately abandon LO jobs in case of resource shortage and this leads to a high percentage of jobs that miss their deadline.

The *Lazy Bailout Protocol* (LBP) is built upon BP and inherits from it the following three execution modes that work as specified below:

1. *Normal*: it is the starting system execution mode. It corresponds to a low-criticality mode where all jobs within the system are supposed to be processed correctly according to the C_{LO} threshold.

2. *Bailout*: it is the emergency mode that is entered whenever a HI job overruns its C_{LO}.

3. *Recovery*: it is the emergency mode that is entered to allow to the last pending lowest priority HI job to complete before to go back to Normal mode.

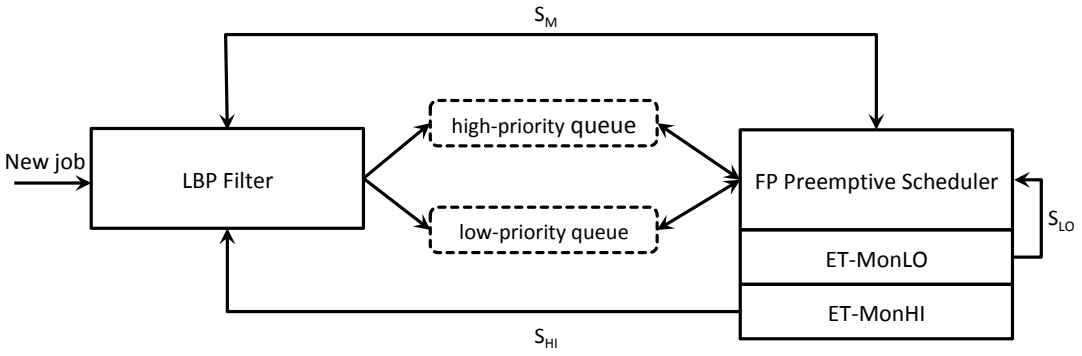


Figure 4.1: LBP architecture

Figure 4.1 shows the components of LBP. The LBP filter is responsible for changing the execution modes. The system has two ready queues for jobs: the *high-priority queue* represents the standard ready jobs queue while the *low-priority queue* keeps the LO jobs that have been released during emergency modes or that have exceeded their C_{LO}. In both queues, jobs are sorted according to *Deadline Monotonic* (DM), i.e., instances of tasks with smallest relative deadlines are considered first. The DM policy has been preferred over the *Rate Monotonic* (RM) since it represents its generalisation and it is still possible to process tasks generated according to the three assumptions introduced in Section 4.1. Furthermore, there are two job monitors to check respectively LO and HI jobs that overrun their C_{LO}. *ET-MonLO* signals to the real-time scheduler the LO jobs that have to be inserted within the low-priority queue while *ET-MonHI* communicates to the LBP filter when a HI job exceeds its optimistic WCET to switch the execution mode to *Bailout*.

LBP inherits from BP the control mechanism that is in charge of the execution mode changes that permits a fast recovery from the emergency modes back to the *Normal* mode. Such mechanism is based on the detection of idle instants and on the value of a decision variable named *Bailout Fund* (BF). The Figure 4.2 shows how the execution mode changes in the scheduling protocol. It contains the events that trigger the switch to a different execution mode together with the related update of the BF value. The system starts in *Normal* mode and then,

if any HI job overruns its C_{LO}, then the BF variable is initialised and there is a change to *Bailout* mode. Once the system is in this mode, the BF variable is updated with the earlier completion of jobs, the release of new LO jobs or the HI jobs overrunning their C_{LO}. If an idle instant occurs, then *Normal* mode is entered whereas if the BF becomes zero then Recovery mode is entered. After the lowest priority pending HI job completes its execution in *Recovery* mode, the system goes back to *Normal* mode.

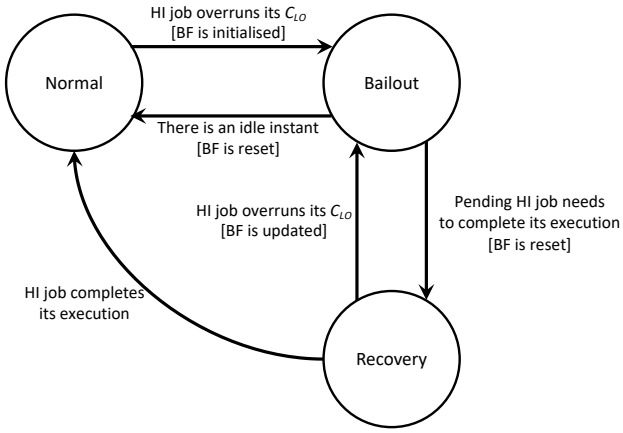


Figure 4.2: Execution mode changes in LBP

The difference between LBP and BP is that LO jobs released in *Bailout* and *Recovery* modes or exceeding their C_{LO} are inserted into the low-priority queue instead of being abandoned. Such jobs run afterwards when the high-priority queue is idle. LO jobs released in *Normal* mode can continue to execute in both *Bailout* and *Recovery* modes and they could even overrun their deadlines as long as they do not exceed their C_{LO}. Below is a detailed description of LBP in each of its execution modes:

Normal mode:

- While all HI jobs execute for no more than their C_{LO} values, the system remains in this mode.
- If any HI job overruns its C_{LO} without signalling completion, then the system switches into the *Bailout* mode and the BF is initialised to $BF = C_{HI} - C_{LO}$.
- LO jobs that overrun their C_{LO} are interrupted and inserted into the low-priority queue.

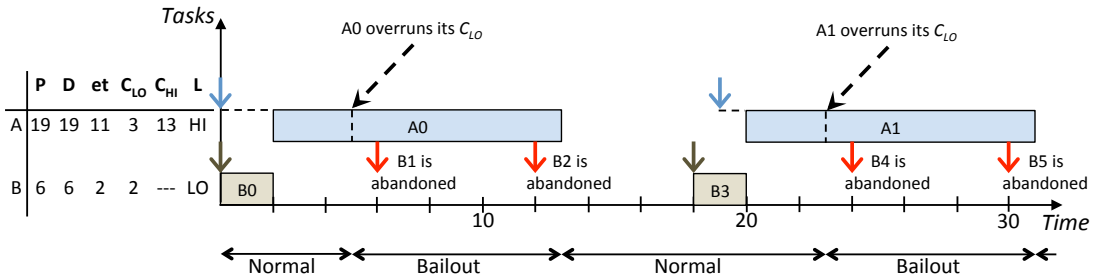
- LO jobs that have been inserted in the low-priority queue are executed during idle instants. If they do not complete within their deadlines, then they are removed from the low-priority queue.

Bailout mode:

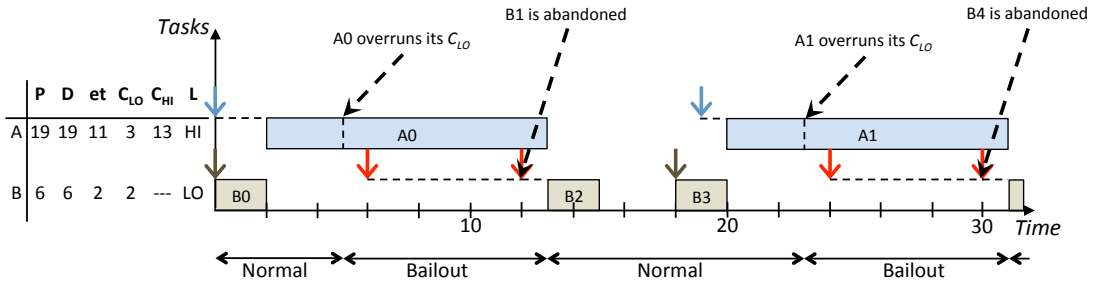
- If any HI job executes for its C_{LO} without signalling completion, then the bailout fund is updated by its maximum extra time budget: $BF = BF + (C_{HI} - C_{LO})$.
- If any HI job completes with an execution time e, with $e \leq C_{LO}$, then its time left is donated to the bailout fund: $BF = BF - (C_{LO} - e)$.
- LO jobs released in *Normal* mode that complete with an execution time of e, with $e \leq C_{LO}$, donate their time left to the bailout fund: $BF = BF - (C_{LO} - e)$.
- If any HI job that already exceeded its C_{LO} completes with an execution time of e, with $C_{LO} < e \leq C_{HI}$, then it donates its extra time left, reducing the bailout fund: $BF = BF - (C_{HI} - e)$.
- LO jobs released in *Bailout* mode are not started but inserted in the low-priority queue to be executed during idle instants in *Normal* mode. Furthermore, when the scheduler would otherwise have dispatched such a job, the job's budget of C_{LO} is donated to the bailout fund: $BF = BF - C_{LO}$.
- If the BF becomes zero, then the lowest priority HI job that did not complete its execution (let this job be j_k) is recorded and the *Recovery* mode is entered.
- If an idle instant occurs, then a transition is made to *Normal* mode, and BF is reset to zero.

Recovery mode:

- LO jobs released in this mode are not started but inserted within the low-priority queue to be executed during idle instants in *Normal* mode.
- If any HI job executes for its C_{LO} value without signalling completion, then the system switches back to *Bailout* mode and BF is initialised: $BF = C_{HI} - C_{LO}$.
- When the job j_k noted at the point when *Recovery* mode was last entered completes, then the system swicthes to *Normal* mode.



(a) BP abandons all LO jobs released in Bailout mode



(b) LBP rescues the LO jobs $B2$ and $B5$ while $B1$ and $B4$ are abandoned after they miss their deadlines

Figure 4.3: Comparison between BP and LBP: LBP schedules more LO jobs than BP

Figure 4.3 shows a mixed-criticality task set that is scheduled according to the standard and the lazy bailout approaches. The task set consists of two tasks, the HI task A and the LO task B. The scheduling starts at time $t = 0$ as soon as the first jobs arrive. As it is possible to notice, jobs are sorted according to the DM priority assignment. Priorities are set to indicate the urgency of a job that in my work is dictated by the relative deadline of the task from which that job is generated. However, the ordering of jobs according to their timing constraints does not reflect the impact their completion has on the system functioning or on the system safety. Such impact is represented by the criticality, indicated with HI and LO in Figure 4.3. At times $t = 5$ and $t = 23$, jobs A_0 and A_1 unexpectedly exceed their C_{LO} but they should be preempted by the higher priority LO jobs generated by task B. Since task A has criticality HI, the completion of its instances has to be guaranteed in all operating conditions. Therefore the optimistic WCET overruns of jobs generated by task A determines the entering of the *Bailout* mode that allows them to complete.

Figure 4.3 shows that the standard BP abandons all the LO jobs released during the *Bailout* modes while LBP allows to recover and schedule more LO jobs. In Figure 4.3.b) jobs B_1 and B_4 are released respectively at times $t = 6$ and $t = 24$ and they have the highest priority. Such jobs are inserted in

the low-priority queue to be removed respectively at times $t = 12$ and $t = 30$ when they miss their deadlines and the next instance of the same task arrives. Furthermore, the LO jobs B_2 and B_5 released respectively at times $t = 12$ and $t = 30$ are executed aftewards in *Normal* mode since there are idle instants to exploit before their deadlines. Such example highlights how LO jobs that are delayed, instead of being abandoned, are executed during idle instants in *Normal* mode to not influence the real-time behaviour of jobs in the high-priority queue. In the whole, compared with LBP, the standard BP results in a decrease of the system utilisation because whenever there is interference among HI and LO jobs released in *Bailout* or *Recovery* modes, then LO jobs are simply abandoned. On the other hand, LBP increases the processor utilisation by exploiting the system idle time and, by doing this, it improves the overall service provided to LO tasks and this is achieved by increasing the number of LO jobs that are processed.

4.3 SLBP: An LBP Variant with Soft Deadlines

The *Soft Lazy Bailout Protocol* (SLBP) represents an extension of LBP to use in cases in which a LO job completion after the deadline still has some utility. This can be the case, for example, for some multimedia and image processing tasks with constrained deadlines. In such systems it can be useful that some jobs are completed even after their deadlines, as long as this happens before the arrival of next frame to process or of the next instance of the same task. In such cases, a late result is still useful and better than no result because it can contribute to keep the service quality provided above a predefined threshold.

SLBP works like LBP with the only difference that LO jobs inserted in the low-priority queue are treated as soft real-time jobs that have a bounded tardiness. The extent of the tardiness for a LO job depends on the amount of system idle time available till the arrival of next instance of the same task. In contrast, with LBP LO jobs not completing within their deadline are aborted and the idle time made available is used to schedule new incoming LO jobs in the low-priority queue. The SLBP protocol deals with LO jobs as follows:

1. If a LO task has a deadline smaller than its period, then SLBP allows to its jobs within the low-priority queue to overrun their deadline till the arrival of next instance of the same task.

2. If a LO task has deadline equal or greater than period, then no deadline overrun is allowed for its instances.

Therefore, any LO job in the low-priority queue with a deadline smaller than period can exploit the idle time between its deadline and the release of its successive instance to complete. From this point of view, SLBP permits LO jobs in the low-priority queue to have a bounded tardiness without interfering with successive instances of same task.

Figure 4.4.a) shows a task set scheduled with LBP while Figure 4.4.b) represents the schedule generated by SLBP. The task set consists of two tasks with constant computing time. Jobs of LO task B always have an execution time of $et = 2$ while instances of task A have a constant execution time of $et = 9$. The first instance of HI task A overruns its C_{LO} at time $t = 9$ and the system switches to *Bailout* mode. The example reveals the difference in how LO jobs within the low-priority queue are scheduled with the two protocols. With LBP, the jobs within the low-priority queue that do not complete within their deadline are abandoned. On the other hand, the SLBP always treats the HI tasks as hard real-time tasks but tries to increase the overall completion rate of LO jobs by allowing to those inserted in the low-priority queue to have a bounded tardiness.

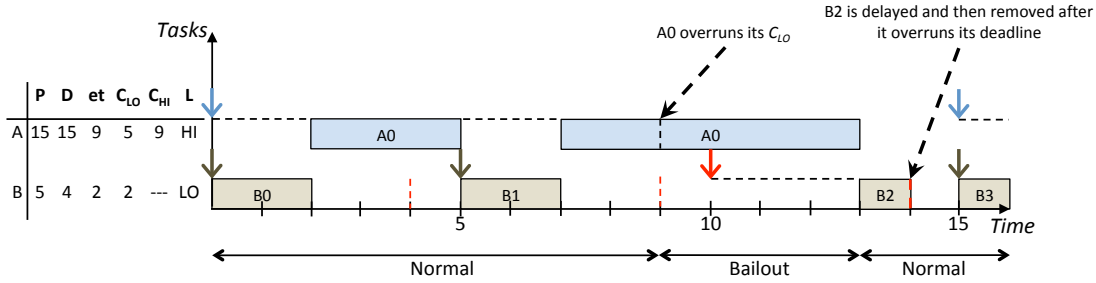
As showed in Figure 4.4a), with LBP, the job B_2 is relased during the Bailout mode at time $t = 10$ and suddenly inserted in the low-priority queue. B_2 starts its execution at time $t = 13$ but it is interrupted and abandoned at time $t = 14$ since it misses its deadline. Conversely, as showed in Figure 4.4b), SLBP allows to B_2 to overrun its deadline and to run till the arrival of next instance of the same task. B_2 completes on time at time $t = 15$.

It is worth to notice that if task B has a deadline equal to its period, then the schedules generated by LBP and SLBP are exactly the same with job B_2 that would complete within its deadline at time $t = 15$ in both cases.

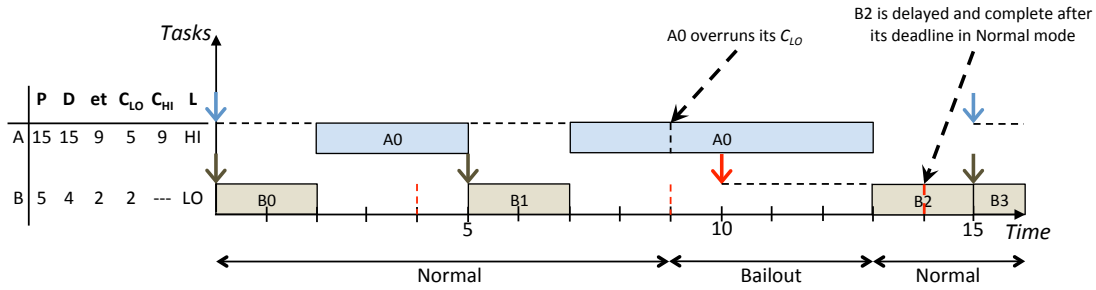
4.4 Integration of LBP and SLBP with Complementary Scheduling Techniques

The LBP and SLBP protocols can be further enhanced by means of scheduling strategy to increase the amount of LO jobs completed within their deadlines.

The first technique uses an offline analysis to check how much it is possible to increase the task set utilisation factor without making the system unschedulable [44, 106, 17]. It increases the worst-case system load till the maximum point for which the task system is still schedulable according to the three assumptions stated in Section 4.1. Since the aim is to increase the amount of LO jobs successfully scheduled, the offline scaling up of the optimistic WCET estimates is



(a) LBP aborts the execution of job B_2 after its deadline



(b) SLBP allows to B_2 to overrun its deadline till the arrival of next instance of same task

Figure 4.4: LBP aborts LO job B_2 while SLBP allows for its late completion.

made only for the HI tasks. This allows to enlarge the duration of the *Normal* mode. This technique consists of the following two steps:

1. The C_{LO} of each HI task is increased by the upper value α^* that still preserves schedulability as follow:

$$\forall \tau_i \in \tau_{HI}. \sum_{i=1}^{k} \alpha^* \cdot \frac{C_{LO,i}}{P_i}$$

2. Then, if possible, the C_{LO} of each individual HI task is enlarged in order of increasing deadline. This is done because some C_{LO} could be still increased without making the system unschedulable.

The scheduling protocols resulting from the integration of this technique with LBP and SLBP are named respectively LBPS and SLBPS.

The second technique is based on the exploitation of the amount of computing time budget of a job that is estimated offline but not required at runtime because of its earlier completion. Such time budget left is usually named *gain time* [40, 41, 17]. The gain time g_i is defined as follows:

$$g_i = et(j_i) - C_{LO}$$

where $et(j_i)$ represents the execution time of job j_i, with $et(j_i) < C_{LO}$. It is worth to notice that the gain time is never negative. The budget g_i is passed from a job to the next lower priority job if there are no idle instants and only during the *Normal* mode. Furthermore, the gain time collection at runtime is made only for jobs in the high-priority queue. The exploitation of the gain time in LBP and SLBP led to devise LBPG and SLBPG protocols.

Finally, by integrating both techniques described above in LBP and SLBP, I also introduced two further refinements that are named LBPSG and SLBPSG.

4.5 Chapter Summary

In this chapter, I introduced the LBP protocol, a BP refinement that allows to further increase the amount of LO jobs scheduled.

Section 4.1 describes the task and system model together with the schedulability assumptions used with the LBP protocol. A detailed description of how LBP works in each of its execution modes is provided in Section 4.2. Section 4.3 introduces and describes the SLBP, an LBP variant that treats LO jobs as soft real-time jobs. With SLBP, LO jobs in the low-priority queue can overrun their deadline as long as they complete before the arrival of next instance of the same task. Hence, LBP allows to LO jobs to run during idle time till they reach their absolute deadline while SLBP permits a reasonably bounded tardy completion. Finally, Section 4.4 shows how to enhance LBP and SLBP with complementary scheduling techniques.

Chapter 5

Formal Comparison of Variants of BP and LBP

This chapter contains a formal evaluation of BP, LBP and their derivatives scheduling protocols. The system model considers set of tasks having two criticality, HI and LO (with HI being more critical than LO), as defined in Chapter 4. HI tasks have two WCET estimates, the former more optimistic and the latter more conservative, indicated respectively with C_{LO} and C_{HI}. Conversely, for LO tasks only the knowledge of easier to derive C_{LO} estimates is assumed.

Section 5.1 introduces a formal criterion to compare different mixed-criticality scheduling methods within hard real-time settings. This criterion considers and compares jobs scheduled at different criticality levels and, for each level, jobs *successfully scheduled* are those completed within their deadlines. The correct completion of tasks with highest criticality is considered first. The best mixed-criticality performance will be that of the scheduling protocol in which the largest amount of highest criticality level jobs are successfully scheduled. With regard to the dual-criticality systems considered, such metrics considers feasibility of HI tasks as most important. If two protocols lead to the same amount of HI jobs completing within their deadlines, then it is also considered the LO tasks schedulability. Section 5.2 assesses the BP protocol with its derivatives (BPG, BPS and BPSG). Section 5.3 compares LBP with the remaining protocols, both BP and LBP based, including SLBP.

The outcome highlights how LBP always schedules more LO jobs than BP and each LBP-based method behaves better than the corresponding BP-based protocol. Finally, due to difference in the system idle-time exploitation, LBP has a better LO tasks success rate than SLBP.

5.1 Predicates and Comparison Criterion

In this section I formalise a criterion to compare two different mixed-criticality systems. Below are defintions and predicates used to prove the theorems afterwards.

STS, τ, JS:
: STS is a set of task sets τ. τ is an individual scheduling problem consisting of tasks. JS is a set of jobs created at runtime by scheduling a task set.

METHOD:
: This is the scheduling method applied, e.g., `BP`, `BPG`, `BPS`, `BPSG` or `LBP`.

`HI`(τ), `LO`(τ): $\tau \to \tau$:
: `HI`(τ) is a subset of τ containing only tasks of HI criticality. `LO`(τ) is a subset of τ containing only tasks of LO criticality.

`Scheduled`(mtd, τ): METHOD $\times\ \tau \to$ JS:
: The job set generated from a task set τ, which is *successfully* scheduled with method mtd, i.e., jobs which completed within their deadline.

`ScheduledHI`(mtd, τ): METHOD $\times\ \tau \to$ JS:
: This includes only those jobs from `Scheduled`(mtd, τ) which are derived from tasks with HI criticality.

`ScheduledLO`(mtd, τ): METHOD $\times\ \tau \to$ JS:
: This includes only those jobs from `Scheduled`(mtd, τ) which are derived from tasks with LO criticality.

`Failed`(mtd, τ): METHOD $\times\ \tau \to$ JS:
: The job set generated from a task set τ, which is *not successfully* scheduled by method mtd, i.e., jobs which were not completed within their deadline.

`Abandoned`(mtd, τ): METHOD $\times\ \tau \to$ JS:
: This predicate returns the set of jobs generated from a task set τ, which were never forwarded by the mixed-criticality scheduling method mtd to its underlying real-time scheduler. This is a special case of failed jobs:

$$Abandoned(mtd, \tau) \subseteq Failed(mtd, \tau)$$

: Abandoned jobs are also different from *dropped jobs*, which are jobs that failed after having started their execution with the underlying real-time scheduler.

`LORated(`mtd, τ`):` METHOD $\times\ \tau \rightarrow$ JS:

This predicate returns the set of LO jobs, which were re-queued from the default high-priority queue to the low-priority queue. This method is not defined for BP and its derivatives.

`IsBetterMCS(`mtd_1, mtd_2, τ`):`

METHOD$^2 \times\ \tau \rightarrow$ BOOL: This predicate tests whether a scheduling method mtd_1 is better than method mtd_2 for a task set τ with respect to mixed-criticality scheduling, which is formally defined as:

$$IsBetterMCS(mtd_1, mtd_2, \tau) \Rightarrow \begin{cases} \text{TRUE if } (ScheduledHI(mtd_1, \tau) \supset \\ \qquad ScheduledHI(mtd_2, \tau)) \vee \\ \qquad (ScheduledHI(mtd_1, \tau) == \\ \qquad ScheduledHI(mtd_2, \tau) \wedge \\ \qquad (ScheduledLO(mtd_1, \tau) \supset \\ \qquad ScheduledLO(mtd_2, \tau))) \\ \text{FALSE otherwise} \end{cases}$$

This tests whether mtd_1 has a better performance than mtd_2 for HI jobs, or equal performance for HI jobs but better performance for LO jobs.

`IsBetterMCS(`mtd_1, mtd_2`):`

METHOD$^2 \rightarrow$ BOOL: This predicate tests whether a scheduling method mtd_1 is better than method mtd_2 for all task sets with respect to mixed-criticality scheduling, which is formally defined as:

$$IsBetterMCS(mtd_1, mtd_2) \Rightarrow \begin{cases} \exists \tau \in \text{STS}.\ IsBetterMCS(mtd_1, mtd_2, \tau) \\ \qquad\qquad \wedge \\ \nexists \tau \in \text{STS}. IsBetterMCS(mtd_2, mtd_1, \tau) \end{cases}$$

It is worthwhile to note that $IsBetterMCS(mtd_1, mtd_2, \tau)$ and $IsBetterMCS(mtd_1, mtd_2)$ are transitive:

$$IsBetterMCS(m_A, m_B)\ \wedge\ IsBetterMCS(m_B, m_C) \Rightarrow IsBetterMCS(m_A, m_C)$$

5.2 Comparison of BP-based protocols

This section compares the standard BP protocol with its variants derived by using the offline sensitivity analysis [44, 17] and the online gain time collection [17].

5.2.1 Comparison between BP and BPG

Theorem 5.2.1 *BPG has the same success rate of HI tasks than BP, which can be formally written as:*

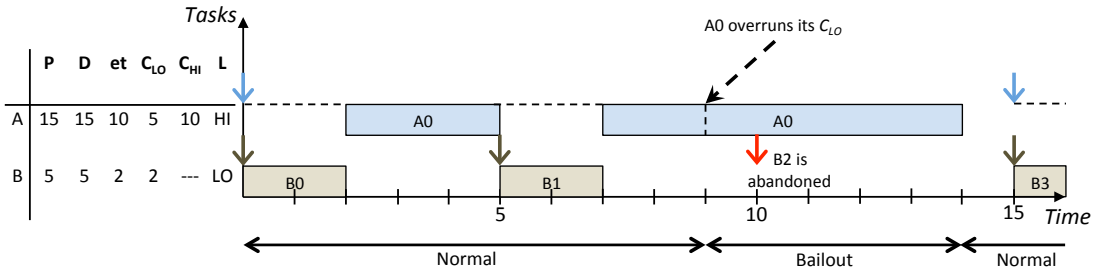
$$\forall\ \tau \in \text{STS}.\ \mathit{ScheduledHI}(BP, \tau) == \mathit{ScheduledHI}(BPG, \tau)$$

Proof (Theorem 5.2.1) BPG differs from BP only for the exploitation of available computing resources not used during execution in *Normal* mode. The online gain time collection made in *Normal* mode does not affect the task sets schedulability [17].

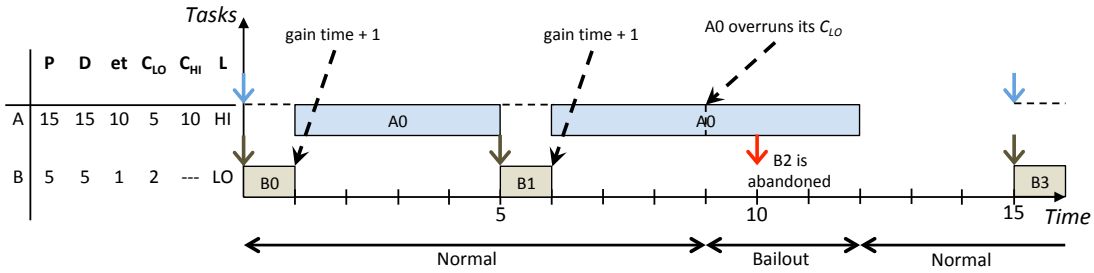
The gain time collection mechanism allows to adjust at runtime the optimistic WCET of jobs. Jobs that complete before their C_{LO} give the amount of time budget left to the next job only when there is no idle time among them. Because of this, the worst-case response time computed according to **Assumption 1** is not increased and the system continues to be schedulable. In fact, the increase of the utilisation of a job for which the optimistic WCET has been increased is compensated by the decrease of the same amount in utilisation of the higher priority job that donated its gain time. As a result, the worst-case system load during *Normal* mode remains constant and the first and third assumptions checked offline are still true. Figure 5.1 shows an example in which it is possible to notice how the gain time collection at runtime does not increase the worst-case response time in *Normal* mode. In particular, Figure 5.1.a) shows the schedule generated by the BP protocol when each LO job uses all its time budget C_{LO} to complete while Figure 5.1.b) shows that generated by BPG with jobs B_0 and B_1 completing earlier than their C_{LO}. By comparing them, it is possible to notice that in both cases the system switches to *Bailout* mode at time $t = 9$. As a consequence, the worst-case response time of the HI task A in Normal mode is not increased and the third assumption continues to guarantee that no HI job misses its deadline.

Therefore BPG has the same performances than BP with regard to HI jobs:

$$\forall \tau \in \text{STS}.\ \mathit{ScheduledHI}(BP, \tau) == \mathit{ScheduledHI}(BPG, \tau)$$



(a) Schedule generated by BP when each LO job execute till its C_{LO}



(b) Schedule generated by BPG when each LO job completes earlier then its C_{LO}

Figure 5.1: (Proof of Theorem 5.1) The gain time collection does not increase the worst-case system load in Normal mode

□

Theorem 5.2.2 *BPG can have a better success rate of LO tasks than BP, but never worse, which can be formally written as:*

$$\forall\ \tau \in \text{STS}.\ \mathit{ScheduledLO}(BP, \tau) \subseteq \mathit{ScheduledLO}(BPG, \tau)$$
$$\exists\ \tau \in \text{STS}.\ \mathit{ScheduledLO}(BP, \tau) \subset \mathit{ScheduledLO}(BPG, \tau)$$

Proof (Theorem 5.2.2) The proof consits of two parts:

1. Showing that for all possible task sets τ it holds that

$$\mathit{ScheduledLO}(BPG, \tau) \supseteq \mathit{ScheduledLO}(BP, \tau)$$

2. Showing by example that there exists a task set τ such that

$$\mathit{ScheduledLO}(BPG, \tau) \supset \mathit{ScheduledLO}(BP, \tau)$$

Part 1:

The gain time collection at runtime allows to reduce the number of times the

system goes in *Bailout* mode and the amount of time the system remains in such mode [17]. The enlargement of the *Normal* mode rescues more LO jobs from being abandoned. Hence, it reduces the amount of LO jobs that do not complete within their deadlines. Therefore BPG can never have worse performances with respect to scheduling LO jobs:

$$\forall \tau \in \text{STS}.\ \mathit{ScheduledLO}(\mathit{BPG}, \tau) \supseteq \mathit{ScheduledLO}(\mathit{BP}, \tau)$$

Part 2:

Figure 5.2 shows the existence of a task set with two tasks having implicit deadlines in which all jobs meet their deadlines if scheduled with BPG while one LO job fails with BP. In the example, jobs of HI task A have an execution time of $et = 3$, except for A_0 for which $et(A_0) = 5$. Furthermore, $C_{LO}(A) = 4$ and $C_{HI}(A) = 7$. The jobs of LO task B have an execution time of $et = 2$, except for B_0 for which $et(B_0) = 1$ (written as $2{-}1$ in Figure 5.2).

As shown in Figure 5.2.b), BPG schedules all HI and LO jobs successfully and never enters into HI criticality mode. However, as shown in Figure 5.2.a), BP fails to schedule the LO job B_1 successfully. In fact, with BP the job A_0 overruns its $C_{LO}(A_0)$ at time $t = 5$ and the scheduler enters *Bailout* mode. The LO job B_1 arrives at time $t = 6$ during the *Bailout* mode. B_1 is abandoned and the bailout fund is decreased but it remains still positive. Then, the system experiences an idle instant and the scheduler switches back into *Normal* mode. In case of BPG the shorter execution of B_0 increments the gain time by 1, which is then added to $C_{LO}(A_0)$. Thus, job A_0 with $et(A_0) = 5$ does not cause an overrun of $C_{LO}(A_0)$, as $C_{LO}(A_0)$ had been increased by a gain time of 1, originating from job B_0. The system never switches to *Bailout* mode and job B_1 can start its execution as soon as it is released.

Hence, the task set τ of this example is an instance fulfilling the property to be shown:

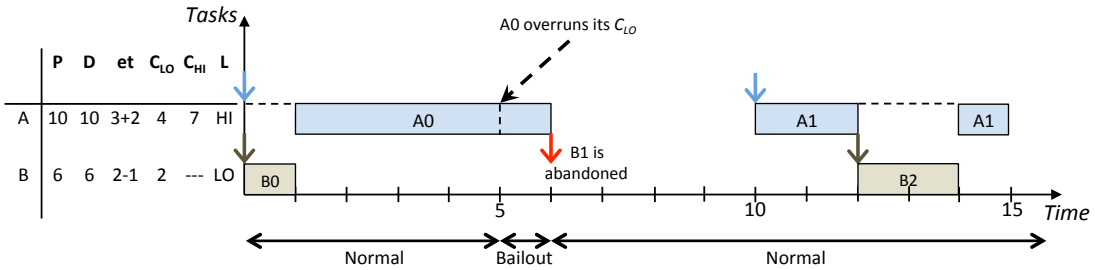
$$\exists\ \tau \in \text{STS}.\mathit{ScheduledLO}(\mathit{BPG}, \tau) \supset \mathit{ScheduledLO}(\mathit{BP}, \tau)$$

□

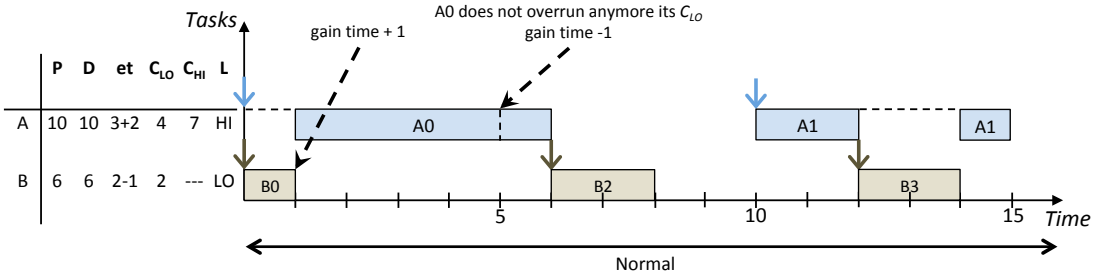
From Theorem 5.2.1 and Theorem 5.2.2 follow that:

Corollary 5.2.3 *BPG has a better mixed-criticality performance than BP, which can be formally written as:*

$$\mathit{IsBetterMCS}(\mathit{BPG}, \mathit{BP})$$



(a) With BP the LO job B_1 is abandoned



(b) With BPG all LO and HI jobs are scheduled successfully

Figure 5.2: (Proof of Theorem 5.2.2) BPG schedules more LO jobs than BP

Proof (Corollary 5.2.3) Theorem 5.2.1 proves that BP and BPG have the same success rate for HI tasks while Theorem 5.2.2 demonstrates that the usage of gain time collection at runtime increases the amount of LO jobs completed within their deadlines. As a result, it follows that:

$$IsBetterMCS(BPG, BP)$$

□

5.2.2 Comparison between BP and BPS

Theorem 5.2.4 *BPS has the same success rate of HI tasks than BP, which can be formally written as:*

$$\forall\ \tau \in \text{STS}.\ ScheduledHI(BPS, \tau) == ScheduledHI(BP, \tau)$$

Proof (Theorem 5.2.4) BP and BPS behave the same way regarding the handling of HI jobs. BPS reduces the number of times the system goes into

Bailout/Recovery mode and enlarges the duration of *Normal* mode by increasing the C_{LO} of HI jobs. The scaling up of C_{LO} increases the system load up to the maximum point for which the task set is still schedulable, i.e., BPS does the upscaling of C_{LO} only as long as the **Assumptions 1 to 3** from Chapter 4 are preserved. Assuming that the schedulability test used to check **Assumptions 1 to 3** is a sufficient test, this concludes the proof that BPS preserves the schedulability of HI tasks. Thus, for any task set τ it follows that

$$ScheduledHI(BPS, \tau) == ScheduledHI(BP, \tau)$$

□

Theorem 5.2.5 *BPS can have a better success rate of LO tasks than BP, but newer worse, which can be formally written as:*

$$\forall\ \tau \in \text{STS}.\ ScheduledLO(BP, \tau) \subseteq ScheduledLO(BPS, \tau)$$
$$\exists\ \tau \in \text{STS}.\ ScheduledLO(BP, \tau) \subset ScheduledLO(BPS, \tau)$$

Proof (Theorem 5.2.5) The proof consists of two parts:

1. Showing that for all possible task sets τ it holds that

$$ScheduledLO(BPS, \tau) \supseteq ScheduledLO(BP, \tau)$$

2. Showing by example that there exists a task set τ such that

$$ScheduledLO(BPS, \tau) \supset ScheduledLO(BP, \tau)$$

Part 1:

Compared with BP, in BPS an offline static analysis is performed to scale up the C_{LO} estimates of all HI tasks as much as possible while preserving the schedulability of the whole task set when considering their optimistic WCET bounds C_{LO}. The scaling of C_{LO} values with BPS reduces the number of times HI jobs overrun their C_{LO}, thus reducing the number of times the *Bailout* mode is entered (since it delays the entering into *Bailout* mode). As a result, BPS may abandon less LO jobs than BP, but never more:

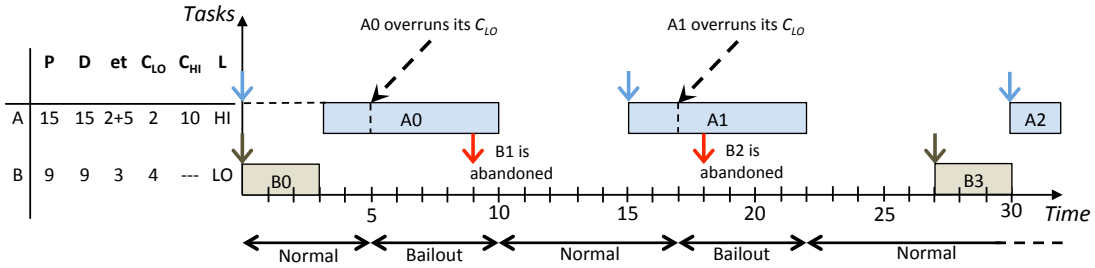
$$\forall\ \tau \in \text{STS}.\ Abandoned(BPS, \tau) \subseteq Abandoned(BP, \tau)$$

This leads to more jobs that can start and successfully complete. Thus, for any task set τ it follows that

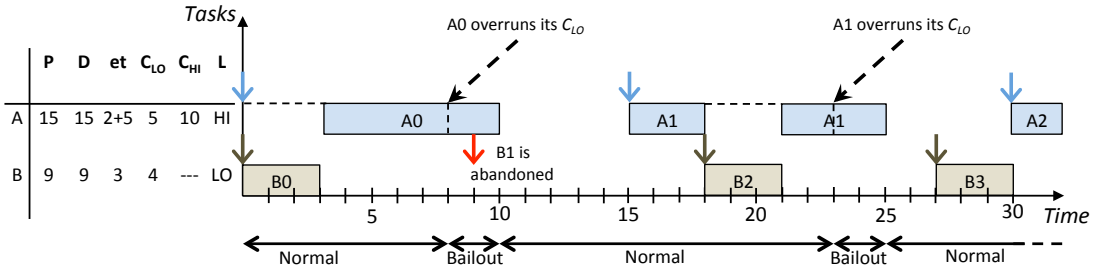
$$ScheduledLO(BP, \tau) \subseteq ScheduledLO(BPS, \tau)$$

Part 2:

Figure 5.3 shows the existence of a task set with tasks having implicit deadlines in which more LO jobs meet their deadlines if scheduled with BPS compared with BP. In the example, jobs of HI task A have an execution time of $et = 2$, except for A_0 and A_1 for which $et(A_0) = et(A_1) = 7$ (denoted as 2+5 in Figure 5.3). Furthermore, task A has $C_{LO}(A) = 2$ in Figure 5.3.a) and $C_{LO}(A) = 5$ in Figure 5.3.b) while its $C_{HI}(A) = 10$. The jobs of LO task B have a constant execution time of $et = 3$.



(a) With BP the jobs B1 and B2 are abandoned



(b) With BPS only the job B1 is abondoned

Figure 5.3: (Proof of Theorem 5.2.5, Part 2) BPS schedules more LO jobs than BP.

As shown in Figure 5.3.a), BP schedules successfully all HI jobs but abandons the LO jobs B_1 and B_2 during *Bailout* mode. However, as shown in Figure 5.3.b), BPS also schedules all HI jobs but only abandons the LO job B_1 due to the delay in entering into *Bailout* mode after the scaling up of C_{LO} of HI task A. This allows to LO job B_2 to start on time and to complete successfully.

Hence, the task set τ of this example is an instance fulfilling the property to

be shown:

$$\exists\ \tau \in \text{STS}.\ \mathit{ScheduledLO}(\mathit{BPS}, \tau) \supset \mathit{ScheduledLO}(\mathit{BP}, \tau)$$

□

Corollary 5.2.6 *BPS has a better mixed-criticality performance than BP, which can be formally written as:*

$$\mathit{IsBetterMCS}(\mathit{BPS}, \mathit{BP})$$

Proof (Cololary 5.2.6) Theorem 5.2.4 demonstrates that both BP and BPS schedule all HI task instances successfully. Furthermore, Theorem 5.2.5 proves that BPS abandons less LO jobs and, by doing this, leads to better success rate for LO tasks. Hence, it follows that

$$\mathit{IsBetterMCS}(\mathit{BPS}, \mathit{BP})$$

□

5.2.3 Comparison between BPG and BPS

Theorem 5.2.7 *BPG can have a better LO jobs success rate than BPS but there also exist cases where BPG results in a worse LO jobs success rate than BPS, which could be formally written as:*

$$\exists\ \tau \in \text{STS}.\mathit{IsBetterMCS}(\mathit{BPG}, \mathit{BPS}, \tau)$$
$$\exists\ \tau \in \text{STS}.\mathit{IsBetterMCS}(\mathit{BPS}, \mathit{BPG}, \tau)$$

Proof (Theorem 5.2.7) The proof has two parts:

1. Showing by example that there exists a task set τ such that

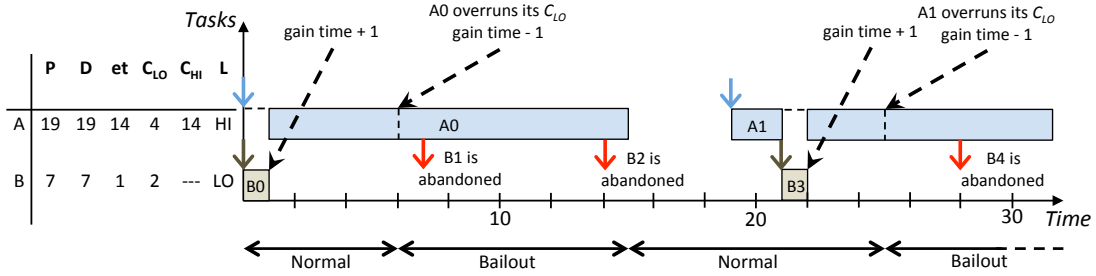
$$\mathit{IsBetterMCS}(\mathit{BPG}, \mathit{BPS}, \tau)$$

2. Showing by example that there exists a task set τ such that

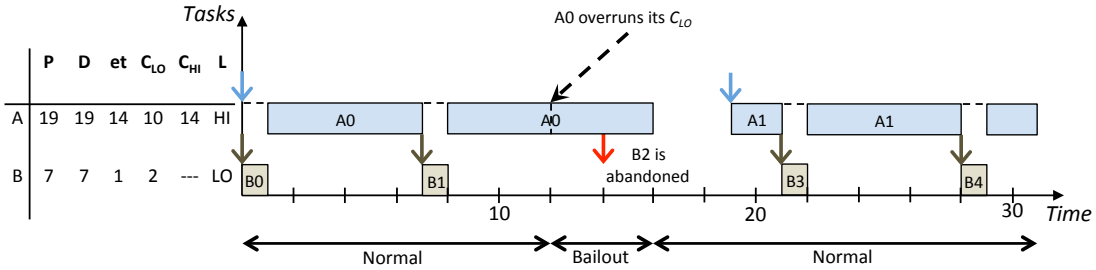
$$\mathit{IsBetterMCS}(\mathit{BPS}, \mathit{BPG}, \tau)$$

Part 1:

Figure 5.4 shows a mixed criticality task set consisting of two tasks having both constant execution times. In particular, the HI task A always has an execution larger than its C_{LO} while the LO task B always has an execution time of $et = 1$ that is smaller than its C_{LO}. The example compares the schedule generated respectively by BPG and BPS. Figure 5.4.a) shows that BPG delays the entering of the *Bailout* mode because the gain time collected from the earlier completion of B instances increases the C_{LO} of HI task A. However, it still abandons jobs B_1, B_2 and B_4. In particular, Figure 5.4.b) represents the BPS schedule. The C_{LO} of the HI task A has been scaled up and the increased duration of the *Normal* mode execution allows to run also jobs B_1 and B_4.



(a) BPG abandons jobs B_1, B_2 and B_4



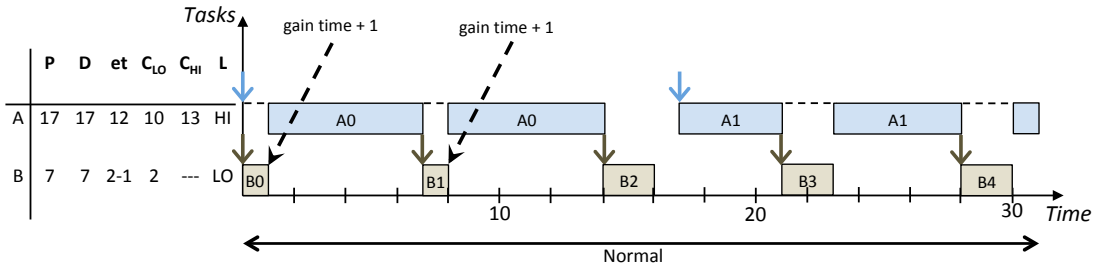
(b) BPS abandons job B_2

Figure 5.4: (Proof of Theorem 5.2.7, Part 1) BPS schedules more LO jobs than BPG

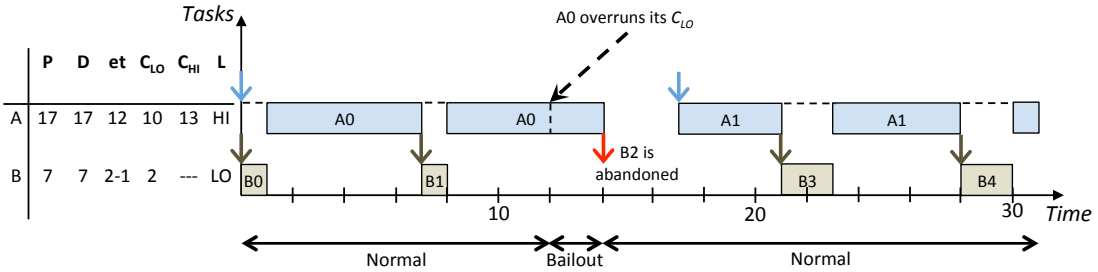
Part 2:

Figure 5.5 shows a task set in which there is no slack time to exploit via sensitivity analysis. In this case, BPS behaves exactly like BP while the BPG online gain time collection allows to rescue more LO jobs. Tha task set consists of two tasks, the HI task A always runs with a constant execution time of $et = 12$ while instances of LO task B can have a variable execution time. In particular, $et(B_0) = et(B_1) = 1$ while $et(B_2) = et(B_3) = et(B_4) = 2$. Furthermore, $C_{LO}(A) = 10$ and $C_{HI}(A) = 13$ while $C_{LO}(B) = 2$. Figure 5.5.a) shows that

the earlier completion of jobs B_0 and B_1 allows to avoid the entering of *Bailout* mode. On the othe hand, Figure 5.5.b) reveals that the HI job A exceeds its C_{LO} at time $t = 12$ and the system switches to *Bailout* mode. The HI job A_0 completes at time $t = 14$ and the bailout fund is reduced but it remains still positive. The LO job B_2 arrives at time $t = 14$ and the bailout fund becomes zero but since the HI job had already completed its execution, then the system goes back to Normal mode.



(a) BPG sucessfully schedules all LO jobs



(b) BPS abandons job B_2

Figure 5.5: (Proof of Theorem 5.2.7, Part 2) BPG schedules more LO jobs than BPS

□

5.2.4 Comparison between BP and BPSG

Corollary 5.2.8 *BPSG has better mixed-criticality performances than BP.*

Proof (Corollary 5.2.8) Corollary 5.2.3 and Corollary 5.2.6 prove that the complementary techniques used together with BP, i.e., the offline scaling up of C_{LO} of HI tasks and the online gain time collection, actually increase the amount of LO jobs that complete within their deadlines without affecting the task set schedulability [17].

Since BPSG results from the integration of the above techniques with the standard BP, it adds the benefits of both to BP. Therefore it follows that:

$$\forall \tau \in \text{STS}.\ \mathit{IsBetterMCS}(\mathit{BPSG}, \mathit{BP})$$

□

5.3 Comparison of LBP with Related Protocols

This section contains the formal comparison of LBP and its variants with the corresponding BP-based scheduling protocols and with SLBP.

5.3.1 Comparison between BP and LBP

Theorem 5.3.1 *LBP has the same success rate of HI tasks than BP, which can be formally written as:*

$$\forall \tau \in \text{STS}.\ \mathit{ScheduledHI}(\mathit{BP}, \tau) == \mathit{ScheduledHI}(\mathit{LBP}, \tau)$$

Proof (Theorem 5.3.1) BP and LBP behave the same way regarding the handling of HI jobs:

1. If a HI job is overrunning its C_{LO}, it is granted an execution budget till C_{HI}.

2. If a HI job does not finish within C_{HI} or within its deadline then it is dropped.

The schedulability of HI tasks is always guaranteed by **Assumptions 1 to 3** introduced in Chapter 4. The only difference between BP and LBP lies in the handling of LO jobs, where LBP puts them in a lower priority scheduling queue instead of abandoning them when released in *Bailout/Recovery* modes or dropping them after the overrun of their C_{LO} as BP does. Since the content of the LBP low-priority queue cannot influence the scheduling of jobs of the default (high-priority) queue, **Assumptions 1 to 3** continue to be valid. It follows that

$$\forall \tau \in \text{STS}.\ \mathit{ScheduledHI}(\mathit{BP}, \tau) == \mathit{ScheduledHI}(\mathit{LBP}, \tau)$$

□

Theorem 5.3.2 *LBP can have a better success rate of LO tasks than BP, but newer worse, which can be formally written as:*

$$\forall\ \tau \in \text{STS}.\ \mathit{ScheduledLO}(BP, \tau) \subseteq \mathit{ScheduledLO}(LBP, \tau)$$
$$\exists\ \tau \in \text{STS}.\ \mathit{ScheduledLO}(BP, \tau) \subset \mathit{ScheduledLO}(LBP, \tau)$$

Proof (Theorem 5.3.2) The only difference between BP and LBP lies in the handling of LO jobs, where LBP puts them in a low-priority scheduling queue instead of abandoning them or dropping them. Hence we have:

$$\forall\ \tau \in \text{STS}.\ \mathit{Abandoned}(BP, \tau) \subseteq \mathit{LORated}(LBP, \tau)$$

Since with BP it happens by definition that

$$\forall\ \tau \in \text{STS}.\ \mathit{Failed}(BP, \tau) \supseteq \mathit{Abandoned}(BP, \tau)$$

to prove Theorem 5.3.2, it only has to be shown that:

$$\exists\ \tau \in \text{STS}.\ \mathit{Abandoned}(BP, \tau) \cap \mathit{LORated}(LBP, \tau) \cap \mathit{Scheduled}(LBP, \tau) \neq \emptyset$$

which means it is sufficient for the proof to show by example that it is possible to have task sets where some LO jobs can be scheduled when the default high-priority queue is idle. To do so, I use the following task set consisting of a HI task A and a LO task B:

Task	P	D	et	C_{LO}	C_{HI}	L
A	15	15	5	3	10	HI
B	4	4	2	2	-	LO

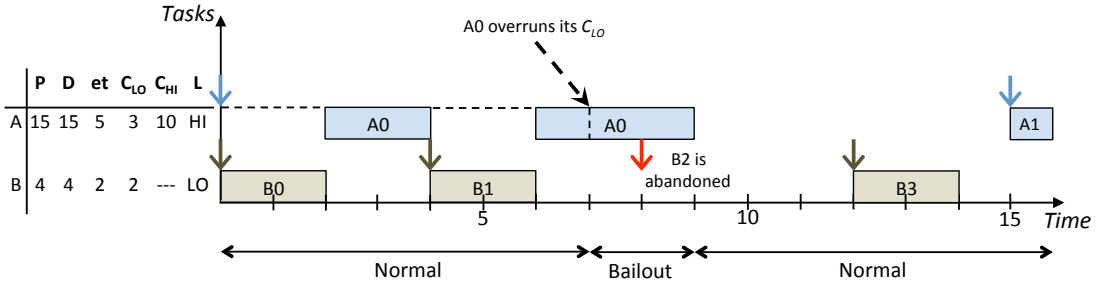
Task A is assumed to have an execution time $et = 5$, which always causes an overrun of the optimistic WCET estimate. The first time, job A_0 exceeds its C_{LO} at $t = 7$ and the system switches into *Bailout* mode. The LO job B_2 is released at time $t = 8$ during *Bailout* mode. Hence, the bailout fund BF is decreased by a quantity equal to the C_{LO} of B_2. However, BF still remains positive. After A_0 completes the system experiences an idle instant and this causes a switch back to *Normal* mode.

As shown in Figure 5.6.a), BP immediately abandons job B_2 at its arrival time during the HI criticality execution mode. In contrast, as shown in Figure 5.6.b), LBP moves such job into the low-priority queue at its arrival and executes it when

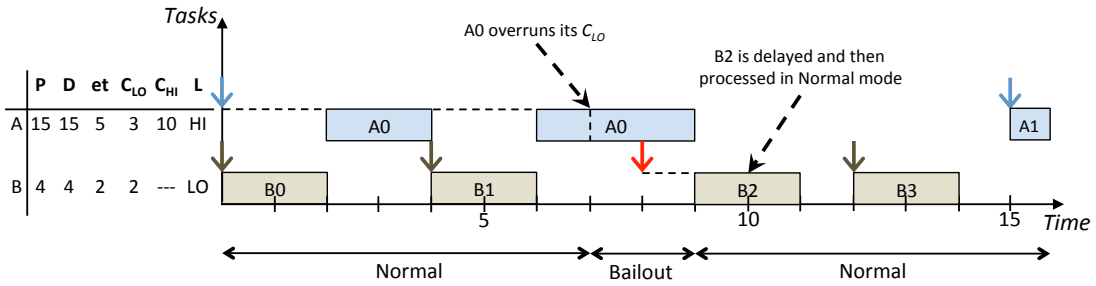
the default queue becomes idle. Thus, this example demonstrates the existence of a task set τ such that

$$\exists\, \tau \in \text{STS}.\ \mathit{Abandoned}(BP, \tau) \cap \mathit{LORated}(LBP, \tau) \cap \mathit{Scheduled}(LBP, \tau) \neq \emptyset$$

which completes the proof. □



(a) BP abandons LO jobs that are not released in Normal mode



(b) LBP provides a delayed execution for job B_2

Figure 5.6: (Proof of Theorem 5.3.2) Example in which LBP successfully executes LO jobs that are abandoned by BP.

From Theorem 5.3.1 and Theorem 5.3.2 it follows that:

Corollary 5.3.3 *LBP has a better mixed-criticality performance than BP, which can be formally written as:*

$$\mathit{IsBetterMCS}(LBP, BP)$$

Proof (Corollary 5.3.3) Theorem 5.3.1 proves that BP and LBP have the same success rate for HI tasks. Furthermore, Theorem 5.3.2 proves that, compared with BP, LBP increases the amount of LO jobs completed within their deadlines. As a result, according to the criterion defined in Section 5.1, it follows that

$$IsBetterMCS(LBP, BP)$$

□

5.3.2 Comparison between BPG and LBPG

Theorem 5.3.4 *LBPG has the same success rate of HI tasks than BPG, which can be formally written as:*

$$\forall \tau \in \text{STS}.\ ScheduledHI(BPG, \tau) == ScheduledHI(LBPG, \tau)$$

Proof (Theorem 5.3.1) BPG and LBPG behave the same way regarding the handling of HI jobs:

1. If a HI job is overrunning its C_{LO}, it is granted an execution budget till C_{HI}.

2. If a HI job does not finish within C_{HI} or within its deadline then it is dropped.

The only difference between BPG and LBPG lies in the handling of LO jobs that exceed their optimistic WCETs or that are released during *Bailout* and *Recovery* modes. BPG abandons such jobs while LBPG inserts them in the low-priority queue for later execution. The content of low-priority queue cannot interfere with the scheduling of jobs within the default queue.

BPG and LBPG schedule all jobs, hence also HI jobs, in the high-priority queue at the same way. Since the gain collection during the *Normal* mode does not alter the worst-case system load, **Assumption 1** guarantees the schedulability of all jobs in such mode. Furthermore, when the system switches to *Bailout* mode, **Assumption 2** and **Assumption 3** guarantee that schedulability of HI tasks is preserved. Thus, it follows that

$$\forall \tau \in \text{STS}.\ ScheduledHI(BPG, \tau) == ScheduledHI(LBPG, \tau)$$

□

Theorem 5.3.5 *LBPG can have a better success rate of LO tasks than BPG,*

but newer worse, which can be formally written as:

$$\forall\ \tau \in \text{STS}.\ \mathit{ScheduledLO}(\mathit{BPG}, \tau) \subseteq \mathit{ScheduledLO}(\mathit{LBPG}, \tau)$$
$$\exists\ \tau \in \text{STS}.\ \mathit{ScheduledLO}(\mathit{BPG}, \tau) \subset \mathit{ScheduledLO}(\mathit{LBPG}, \tau)$$

Proof (Theorem 5.3.5) The only difference between BPG and LBPG lies in the handling of LO jobs, where LBPG puts them in a low-priority scheduling queue instead of abandoning them immediately or dropping them after the overrun of their C_{LO}. Hence we have:

$$\forall\ \tau \in \text{STS}.\ \mathit{Abandoned}(\mathit{BPG}, \tau) \subseteq \mathit{LORated}(\mathit{LBPG}, \tau)$$

Since with BGP it happens by definition that

$$\forall\ \tau \in \text{STS}.\ \mathit{Failed}(\mathit{BPG}, \tau) \supseteq \mathit{Abandoned}(\mathit{BPG}, \tau)$$

to prove Theorem 5.3.5 it is necessary to show that:

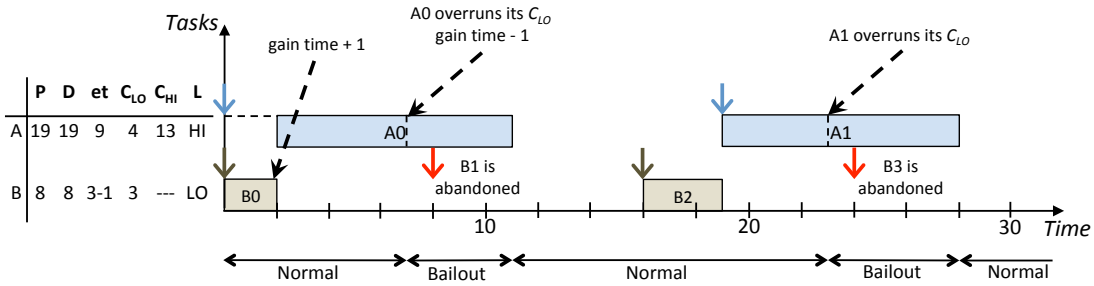
$$\exists\ \tau \in \text{STS}.\ \mathit{Abandoned}(\mathit{BPG}, \tau) \cap \mathit{LORated}(\mathit{LBPG}, \tau) \cap \mathit{Scheduled}(\mathit{LBPG}, \tau) \neq \emptyset$$

Figure 5.7 shows an example of task set in which LBPG rescues LO jobs by moving them into the low-priority queue instead of abandoning them. The instances of HI task A have a constant execution time $et = 9$, which always causes an overrun of the C_{LO} estimate. On the other hand, jobs of LO task B always have an exeuction time of $et = 3$ apart from B_0 that runs only for 2 time units, which allows to have a gain time of 1. Job B_0 completes earlier at time $t = 2$ and gives its gain time to job A_0 for which the optimistic time budget is now updated to 5. A_0 enters the *Bailout* mode at time $t = 7$ and then runs till its completion. No other gain time is collected in this example. Figure 5.7.a) and Figure 5.7.b) show respectively that BPG abandones job B_1 and B_3 while LBPG runs them in *Normal* mode during idle time. Thus, this example demonstrates the existence of a task set τ such that

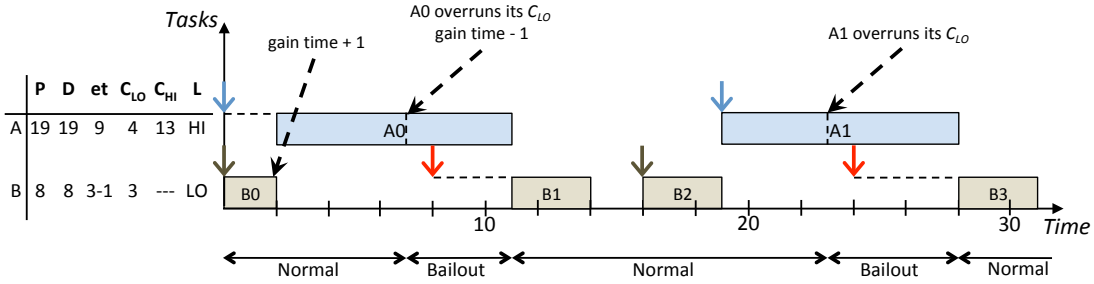
$$\exists\ \tau \in \text{STS}.\ \mathit{Abandoned}(\mathit{BPG}, \tau) \cap \mathit{LORated}(\mathit{LBPG}, \tau) \cap \mathit{Scheduled}(\mathit{LBPG}, \tau) \neq \emptyset$$

which completes the proof. □

Corollary 5.3.6 *LBPG has a better mixed-criticality performance than BPG, which can be formally written as:*



(a) BPG abandons LO jobs B_1 and B_3 during Bailout mode



(b) LBPG provides a delayed execution for job B_1 and B_3

Figure 5.7: (Proof of Theorem 5.3.5) Example in which LBPG successfully schedules LO jobs that are abandoned by BPG.

$$IsBetterMCS(LBPG, BPG)$$

Proof (Corollary 5.3.6) Theorem 5.3.4 demonstrates that both BPG and LBPG always allow to schedule all HI jobs within their deadlines. Furthermore, Theorem 5.3.5 proves that LBPG increases the amount of LO jobs that complete within their deadlines with respect to BPG. It follows that

$$IsBetterMCS(LBPG, BPG)$$

□

5.3.3 Comparison between BPS and LBPS

Theorem 5.3.7 *LBPS has the same success rate of HI tasks than BPS, which can formally written as*

$$\forall \tau \in \text{STS}.\ ScheduledHI(BPG, \tau) == ScheduledHI(LBPG, \tau)$$

Proof (Theorem 5.3.7) BPS and LBPS behave the same way regarding the handling of HI jobs:

1. If a HI job is overrunning its C_{LO}, it is granted an execution budget till C_{HI}.
2. If a HI job does not finish within C_{HI} or within its deadline then it is dropped.

The only difference between BPS and LBPS lies in the handling of LO jobs that exceed their optimistic WCETs or that are released during *Bailout* and *Recovery* modes. BPS abandons such jobs while LBPS inserts them in the low-priority queue for later execution. The content of low-priority queue cannot interfere with the scheduling of jobs within the default queue.

As a result, **Assumption 1 to 3** guarantee the schedulability of HI tasks in every execution mode. Therefore, it follows that

$$\forall \tau \in \text{STS}.\ \mathit{ScheduledHI}(\mathit{BPS}, \tau) == \mathit{ScheduledHI}(\mathit{LBPS}, \tau)$$

□

Theorem 5.3.8 *LBPS can have a better success rate of LO tasks than BPS, but newer worse, which can be formally written as:*

$$\forall\ \tau \in \text{STS}.\ \mathit{ScheduledLO}(\mathit{BPS}, \tau) \subseteq \mathit{ScheduledLO}(\mathit{LBPS}, \tau)$$
$$\exists\ \tau \in \text{STS}.\ \mathit{ScheduledLO}(\mathit{BPS}, \tau) \subset \mathit{ScheduledLO}(\mathit{LBPS}, \tau)$$

Proof (Theorem 5.3.8) The only difference between BPS and LBPS is in the handling of LO jobs exceeding their C_{LO} or released in *Bailout*/*Recovery* modes, i.e., BPS suddenly abandons them while LBPS inserts them in a low-priority queue for later execution during system idle instants. Therefore, it follows that

$$\forall \tau \in \text{STS}.\ \mathit{Abandoned}(\mathit{BPS}, \tau) \subseteq \mathit{LORated}(\mathit{LBPS}, \tau)$$

Since, with BPS, the amount of LO jobs not completed within their deadlines could be greater than those that are abandoned, it follows that:

$$\forall \tau \in \text{STS}.\ \mathit{Failed}(\mathit{BPS}, \tau) \supseteq \mathit{Abandoned}(\mathit{BPS}, \tau)$$

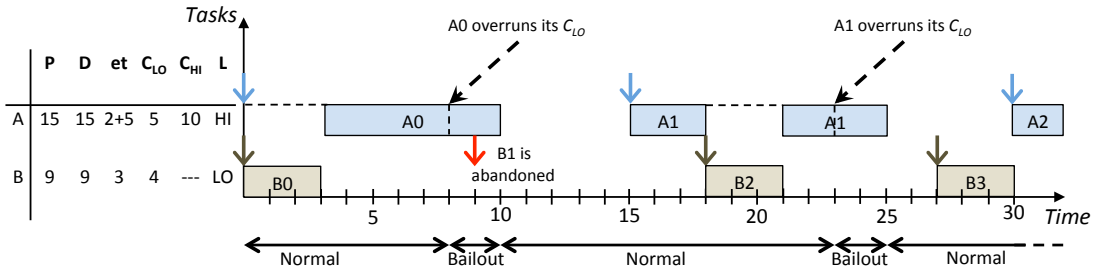
Therefore, to prove that LBPS has a better success rate of LO tasks than BPS, it is necessary to show an example in which LBPS allows to schedule LO

jobs within their deadlines while BPS abandons them, which could be formally written as

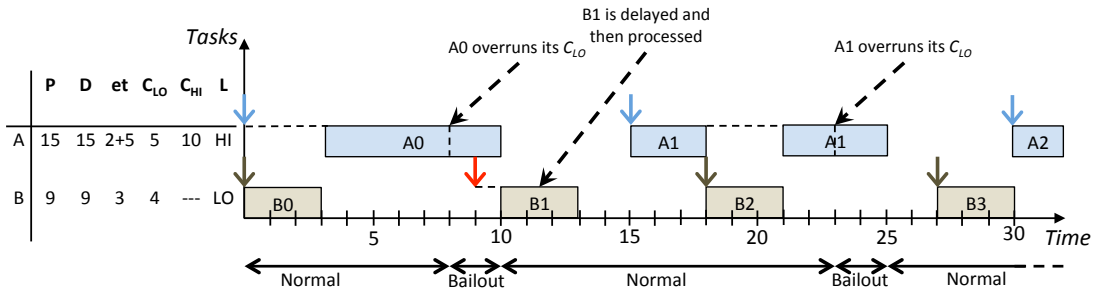
$$\exists\, \tau \in \text{STS}.\ \mathit{Abandoned}(BPS, \tau) \cap \mathit{LORated}(LBPS, \tau) \cap \mathit{Scheduled}(LBPS, \tau) \neq \emptyset$$

Figure 5.8 shows that LBPS has a better LO jobs success rate than BPS. The example shows a task set in which the C_{LO} of HI task A is already scaled up by sensitivity analysis. This enlarges the *Normal* mode execution duration. However, BPS still abandons LO jobs released during HI criticality execution modes. Conversely, LBPS runs them afterwards during idle instants. Figure 5.8 displays how the LO job B_1 released at time $t = 9$ is abandoned with BPS and executed later at time $t = 10$ with LBPS.

This is an instance proving Theorem 5.3.8.



(a) BPS abandons job B_1



(b) LBPS schedules all LO jobs

Figure 5.8: (Proof of Theorem 5.3.9) LBPS schedules more jobs than BPS by provided a delayed execution for LO jobs relased in Bailout mode

□

Theorem 5.3.9 *LBPS has better mixed-criticality performance than BPS, which can be formally written as:*

$$\mathit{IsBetterMCS}(LBPS, BPS)$$

Proof (Theorem 5.3.9) Theorem 5.3.7 proves that both BPS and LBPS schedule successfully every HI tasks. Moreover, Theorem 5.3.8 demonstrates how LBPS increases the amount of LO jobs completed within their deadlines. As a result, it follows that

$$IsBetterMCS(LBPS, BPS)$$

□

5.3.4 Comparison between BPSG and LBPSG

Corollary 5.3.10 *LBPSG has a better mixed-criticality performance than BPSG, which can be formally written as:*

$$IsBetterMCS(LBPSG, BPSG)$$

Proof (Corollary 5.3.10) Theorem 5.3.3 demonstrates that LBP has a better success rate of LO tasks than BP. Moreover, Theorem 5.3.9 and Corollary 5.3.6 prove respectively that the offline scaling up of C_{LO} of HI tasks as well as the online gain time collection mechanism further increases the number of LO jobs completed within their deadlines. As a consequence, it follows that

$$IsBetterMCS(LBPSG, BPSG)$$

□

5.3.5 Comparison between LBP and BPS

Theorem 5.3.11 *LBP can have a better LO jobs success rate than BPS but also exist cases where LBP results in a worse LO jobs success rate than BPS, which could be formally written as:*

$$\exists\, \tau \in \text{STS}.IsBetterMCS(LBP, BPS, \tau)$$
$$\wedge$$
$$\exists\, \tau \in \text{STS}.IsBetterMCS(BPS, LBP, \tau)$$

Proof (Theorem 5.3.11) The proof has two parts:

1. Showing by example that there exists a task set τ such that

$$IsBetterMCS(LBP, BPS, \tau)$$

2. Showing by example that there exists a task set τ such that

$$IsBetterMCS(BPS, LBP, \tau)$$

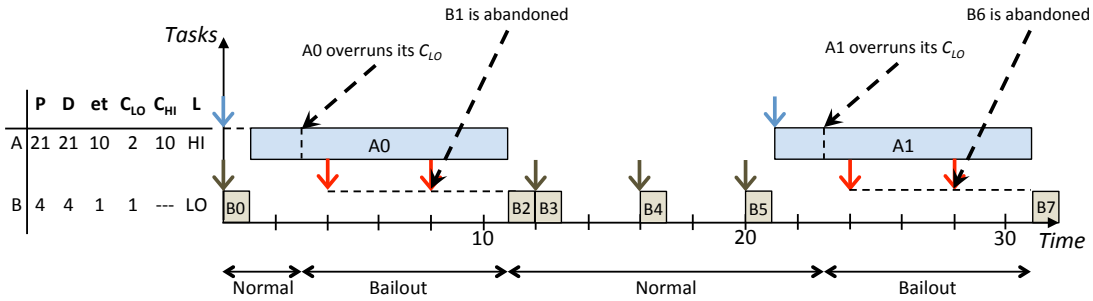
Part 1:

To prove the first part of such theorem I use a task set in which LBP fails in scheduling some LO jobs while BPS, thanks to the scaling up of C_{LO} of the HI task, is able to successfully schedule all LO jobs. The task set consists of two tasks, one HI task with a long execution time and a low arrival rate and a LO task with a short execution time and a higher arrival rate. Figure 5.9.a) shows the LBP execution. The system enters in *Bailout* mode when the HI jobs A_0 and A_1 exceed their C_{LO}, respectively at times $t = 3$ and $t = 23$. The LO jobs released during the *Bailout* mode execution are not abandoned but inserted in the low-priority queue. In particular, jobs B_1 and B_6 are released at times $t = 4$ and $t = 24$ and removed from the low-priority queue at times $t = 8$ and $t = 28$ when next instances of the same task is released. The LO jobs B_2 and B_7 are scheduled during the system idle instants in *Normal* mode.

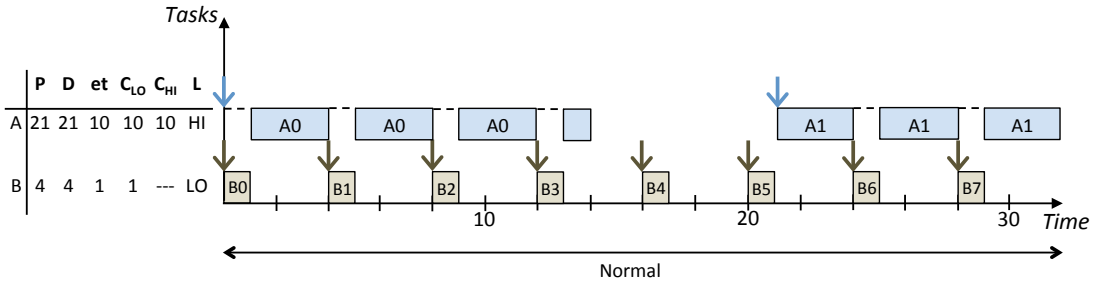
On the other hand, Figure 5.9.b) shows how the scaling up of C_{LO} allows to completely avoid the switch to *Bailout* mode. The system always runs in *Normal* mode and all instances of LO task B are processed as soon as they are relased since they have higher scheduling priority.

Part 2:

The second part of such theorem is proved by showing the existence of a task set in which LBP schedules more LO jobs than BPS. Figure 5.10 shows the existence of a task set that is first processed by LBP and the by BPS. In Figure 5.10.a) LBP schedules more jobs than BPS. The LO jobs B_1 and B_2 are released respectively at times $t = 9$ and $t = 18$ during *Bailout* mode and immediately inserted within the low-priority queue. Then, they are run afterwards during the *Normal* mode. The Figure 5.10.b) shows the BPS execution. In this case, the C_{LO} of the HI task A has been scaled up via sensitivity analysis. This extends the time the system runs in *Normal* mode and, as a result, B_2 is not abandoned since it is now released during the LO criticality mode. However, BPS still drops job B_1.



(a) LBP abandons $B1$ and $B6$ since they start after the arrival of next instance of same task



(b) BPS schedules successfully all LO jobs

Figure 5.9: (Proof of Theorem 5.3.11, Part 1) BPS schedules more LO jobs than LBP

These two examples represent instances proving Theorem 5.3.11.

□

5.3.6 Comparison between LBP and SLBP

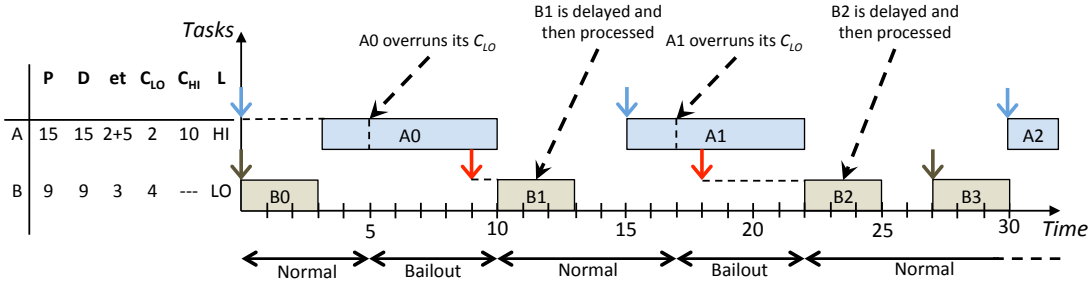
Theorem 5.3.12 *LBP has the same success rate of HI tasks than SLBP, which can be formally written as:*

$$\forall \tau \in \text{STS}.\ \mathit{ScheduledHI}(\mathit{LBP}, \tau) == \mathit{ScheduledHI}(\mathit{SLBP}, \tau)$$

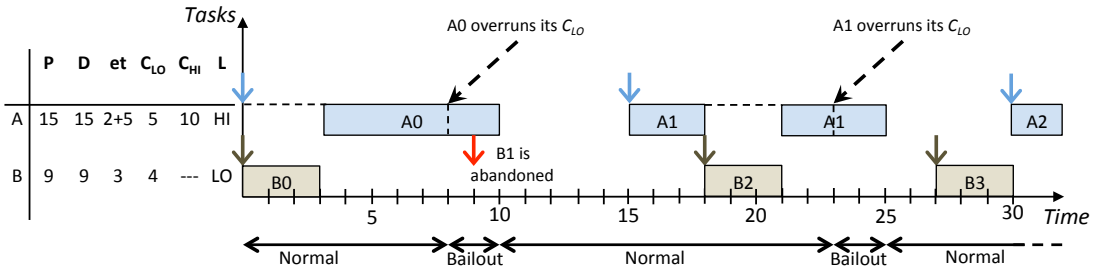
Proof (Theorem 5.3.12) LBP and SLBP behave the same way regarding the handling of HI jobs:

1. If a HI job is overrunning its C_{LO}, it is granted an execution budget till C_{HI}.

2. If a HI job does not finish within C_{HI} or within its deadline then it is dropped.



(a) LBP schedules LO jobs released in Bailout mode during the Normal mode



(b) BPS abandons job B_1

Figure 5.10: (Proof of Theorem 5.3.11, Part 2) LBP schedules more LO jobs than BPS

The schedulability of HI tasks is assured in each execution mode by means of **Assumptions 1 to 3**. The only difference between LBP and SLBP lies in the handling of jobs in the low-priority queue. However, the low-priority queue does not contain HI jobs and it cannot interfere with the scheduling of jobs in the default high-priority queue. Therefore, the feasibilty assumptions continue to be true in every execution modes. It follows that

$$\forall \tau \in \text{STS}.\ \mathit{ScheduledHI}(\mathit{LBP}, \tau) == \mathit{ScheduledHI}(\mathit{SLBP}, \tau)$$

□

Theorem 5.3.13 *LBP can have a better success rate of LO tasks than SLBP, but newer worse, which can be formally written as:*

$$\forall\ \tau \in \text{STS}.\ \mathit{ScheduledLO}(\mathit{SLBP}, \tau) \subseteq \mathit{ScheduledLO}(\mathit{LBP}, \tau)$$
$$\exists\ \tau \in \text{STS}.\ \mathit{ScheduledLO}(\mathit{SLBP}, \tau) \subset \mathit{ScheduledLO}(\mathit{LBP}, \tau)$$

(Note that LO jobs successfully scheduled with *ScheduledLO* are those that terminate within their deadline and not those that complete later since this

chapter provides an evaluation in hard real-time settings. Applications for which a tardy completion of LO jobs is still beneficial should consider other types of metrics.)

Proof (Theorem 5.3.13) The only difference between LBP and SLBP lies in the handling of jobs in the low-priority queue that exceed their deadlines. LBP removes them from the queue and makes room to incoming lower priority LO jobs that might complete within their deadlines. Conversely, SLBP lets LO jobs overrun their deadline to complete as long as they terminate within the arrival of next instance of the same task. As a result, LBP can successfully schedule at least as many LO jobs as SLBP, but never less. This could be formally written as

$$\forall \tau \in \text{STS}.\ \mathit{ScheduledLO}(\mathit{SLBP}, \tau) \subseteq \mathit{ScheduledLO}(\mathit{LBP}, \tau)$$

Figure 5.11 shows an example in which LBP permits to more LO jobs to complete within their deadlines during system idle time. Figure 5.11.a) reveals how the discard of LO jobs that overrun their deadlines allows to successive lower priority jobs in the low-priority queue to exploit the system idle time to sucessfully complete. The jobs C_2, C_3 and B_2 are released during the *Bailout* mode and hence inserted in the low-priority queue to be processed during idle instants in *Normal* mode. Then, C_2 is discarded at time $t = 12$ when it misses its deadline while the remaining jobs run in *Normal* mode. The job C_3 starts at time $t = 16$ and misses its deadline at time $t = 17$ when it is removed from the queue. This allows to the lower priority job B_2 to start its execution and complete before the arrival of job A_1 from the high-priority queue.

Figure 5.11.b) instead highlights how SLBP allows to job C_3 for a tardy completion. However, this prevent execution of the lower priority job B_2 that misses its deadline and it is then abandoned at time $t = 21$ at the arrival of job B_3.

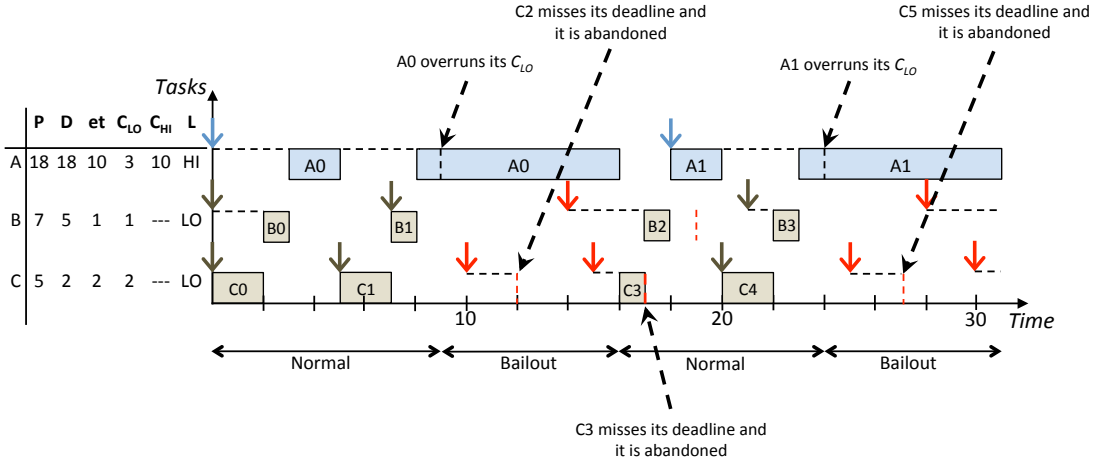
This proves that

$$\exists\ \tau \in \text{STS}.\ \mathit{ScheduledLO}(\mathit{SLBP}, \tau) \subset \mathit{ScheduledLO}(\mathit{LBP}, \tau)$$

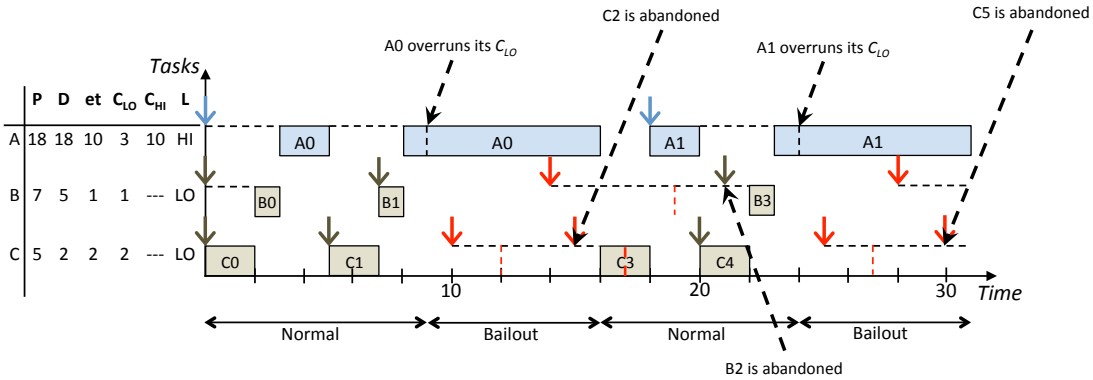
□

Corollary 5.3.14 *LBP has a better mixed-criticality performance than SLBP, which can be formally written as:*

$$\mathit{IsBetterMCS}(\mathit{LBP}, \mathit{SLBP})$$



(a) LBP abandons job C_3 after it misses its deadline and B_2 completes successfully its execution



(b) SLBP allows to job C_3 to complete its execution after its deadline but B_2 cannot execute because of lack of idle time

Figure 5.11: (Proof of Theorem 5.3.13) LBP schedules more LO jobs than SLBP

Proof (Corollary 5.3.14) Theorem 5.3.12 proves both LBP and SLBP always schedule every HI jobs within their deadlines. Moreover, Theorem 5.3.13 demonstrates that LBP can have better success rate of LO tasks than SLBP. Hence, it follows that

$$IsBetterMCS(LBP, SLBP)$$

□

5.4 Chapter Summary

In this chapter I have introduced the predicates and a strategy to evaluate performances of different mixed-criticality scheduling protocols. In particular, the predicate `IsBetterMCS`(mtd_1, mtd_2) allows to compare the performance of two mixed-criticality scheduling methods with priority given to HI jobs that complete within their deadlines. Since, by assumptions, the HI jobs always have to meet their deadlines, the enhancement in scheduling performances is measured by considering the increase in the amount of LO jobs that meet their deadlines.

Based on this criterion, this chapter contains a formal assessment among mixed-criticality methods in hard real-time settings, i.e. jobs *successfully scheduled* are those completed within their deadlines. In particular, Section 5.2 studies mixed-criticality performances of BP-based protocols while Section 5.3 compares LBP with the remaining protocols, including SLBP.

I proved that each LBP-based protocol always increases the amount of LO instances that complete within their deadlines with respect to their corresponding BP-based protocol. On the other hand, no definitive result there is for the comparison between LBP/BPS and BPG/BPS. Lastly, I also showed how LBP allows to process more LO jobs within their deadlines if compared with its variant SLBP.

Chapter 6

The Adaptive Tolerance-based Mixed-Criticality Protocol

This chapter introduces the *Adaptive Tolerance-based Mixed-criticality Protocol* (ATMP), a criticality and utility-aware partitioned scheduling heuristics that maximises the utility on each processing element of a multi-core platform by adjusting the throughput of tasks. The partitioned policy has been preferred to the global scheduling because it avoids the potential excessive overhead due to manipulating a single global queue for all cores, it allows to reuse all the existing scheduling algorithms on each single core and it is supported by the automotive industry, e.g. AUTOSAR [107].

ATMP uses the TRTCM model [22, 1] in which each task can exploit a specific tolerance interval to degrade its service level, measured by means of a utility function. The utility maximisation is made according to the linear programming problem formulated in [6]. The ATMP protocol performs first a reallocation of tasks and then an optimisation of their arrival times in case of sudden unavailability of some core. The schedulability of each optimised set of tasks on each core is guaranteed with an appropriate test. Since the arrival rates optimisation is independent of the underlying scheduling algorithm used to process tasks, any schedulability test can be used to check the task set schedulability on each core. However, we use AMC-rtb [69] that requires the existence of at maximum two WCET estimates per task that indicate respectively the lowest and highest level of assurance required for a correct completion.

Section 6.1 introduces the system and task model together with a description of the TRTCM utility functions. Section 6.2 describes the ATMP optimisation method while Section 6.3 describes the benefits with regard to system safety deriving from the ATMP adoption. Section 6.4 concludes and summarises the

chapter.

6.1 System Model

This section describes the tolerance-based mixed-criticality task model. It is assumed a mixed-criticality system, which consists of multiple tasks that could have different levels of criticality. Each task τ_i of a task set τ is defined as follows:

$$\tau_i = \langle P_i, D_i, \vec{C}_i, L_i, U_i \rangle \tag{6.1}$$

P_i represents the period of a task τ_i, i.e., the inverse of its arrival rate.

D_i is the relative deadline of task τ_i. Task sets have implicit deadlines, i.e., $D_i = P_i$. It is worth to note that such assumption of implicit deadlines has only been chosen for a concrete scheduling test in the implementation, but it is not a requirement of ATMP.

L_i is the criticality level of task τ_i with $L_i > 0$. A higher value of L_i means a higher level of criticality. The vector $\vec{L}$ is used to represent all possible criticality levels in a system: $\vec{L} = (L_1, \ldots, L_k)$, with L_1 being the least and L_k being the maximum criticality level, i.e., $L_1 < L_2 < \ldots < L_k$.

U_i is the relative utility of task τ_i with $0 \leq U_i \leq 1$. The value of U_i is described by a utility function as in Figure 6.1 and varies according to the period of a task τ_i. An *absolute utility* U_i^* is also defined, which is calculated as $U_i^* = U_i \cdot L_i$.

$\vec{C}$ is a vector of at maximum two WCET estimates, indicated respectively by C_{LO} and C_{HI} with $C_{LO} < C_{HI}$. The smaller value C_{LO} represents the upper bound for the level of assurance required at the lower task criticality level while C_{HI} represents the estimate trustworthy at the higher level of assurance required.

The individual instances of a task τ_i at runtime are called jobs. A job j_i is described by the following tuple:

$$j_i = \langle A_i, P_i, D_i, et_{j_i}, \vec{C}, L_i \rangle$$

where A_i is the arrival time and et_{j_i} is the actual execution time. The entries $P_i, D_i, \vec{C}$ and L_i are inherited from the task structure.

6.1.1 Utility Function

The service level provided to each task is measured by means of a utility function and is linked to one or more tasks' parameters. The specific utility-based optimisation performed binds the utility value together with the task periods and permits to adjust the overall system utility by changing the arrival rates of tasks within a task set according the constraints specified in the TRTCM linear programming problem formulated in [6].

The key concept underlying the fault-tolerant optimisation of tasks at runtime is that to tune the task set workload according to the utility and criticality requirements of each task such that the overall system utility is maximised. Such optimisation is made by exploiting the tolerance interval of each task assigned to a specific processing element. Therefore, within the tolerance-based mixed-criticality model the period P_i of a task τ_i is not a given constant, but is assigned by the optimisation method within a certain interval that is specific for each individual task. To be able to calculate the utility of a task, the period P_i of each task $\tau_i \in \tau$ is connected to the following additional utility parameters up_i:

$$up_i = \langle P_{prim,i}, P_{tol,i}, U_{tol,i} \rangle \tag{6.2}$$

$P_{prim,i}$ is the *primary period* of task τ_i, representing the optimal execution rate. For any period $P \leq P_{prim,i}$ the relative utility is one: $u_i = 1$.

$P_{tol,i}$ is the *tolerance period* of task τ_i, which is the maximum period still tolerable for task τ_i.

$U_{tol,i}$ is the *tolerance utility* of task τ_i, which is the relative utility at period $P_{tol,i}$.

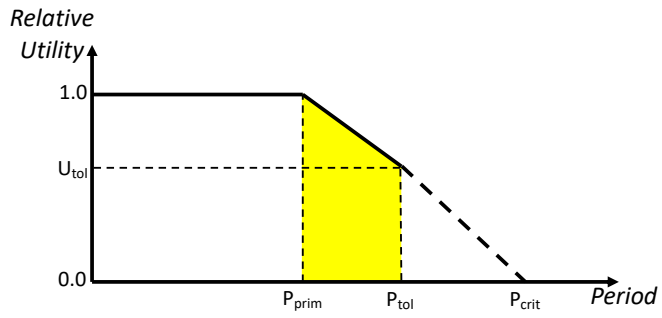


Figure 6.1: Utility function for relative utility calculation based on chosen period

Figure 6.1 shows how the utility parameters up_i describe the utility function of a task τ_i. The modelled tolerance section of the utility function is of linear shape and it is used to smoothly adjust, i.e., degrade or speed up according to the circumstances, the task arrival rates at runtime. Figure 6.1 also contains the *critical period* (indicated as p_{crit}) that represents the arrival rate for which the task utility becomes zero. More details about the tolerance-based real-time model can be found in [6].

The usage of the tolerance range allows to tune the period of each task within its related utility range. Because of this, the adjustment of the utilisation factor of a task corresponds to the adjustment of its related utility value. In the TRTCM task model, the possible load of a task τ_i varies within its so-called primary load $load_{prim,i}$ and its tolerance load $load_{tol,i}$, respectively defined as below:

$$load_{prim,i} = \frac{C_{LO,i}}{P_{prim,i}}, \quad load_{tol,i} = \frac{C_{LO,i}}{P_{tol,i}} \tag{6.3}$$

where $C_{LO,i}$ represents the non-conservative WCET estimate of an individual task τ_i. Consequently, the total system load can be adjusted within $load_{prim}$ and $load_{tol}$ of the whole task set:

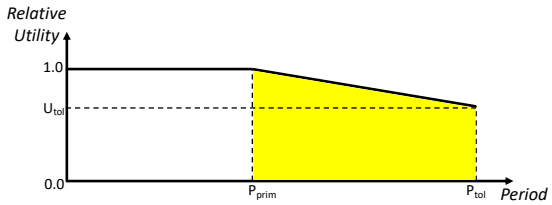
$$load_{prim} = \sum_{\tau_i \in \tau} \frac{C_{LO,i}}{P_{prim,i}}, \quad load_{tol} = \sum_{\tau_i \in \tau} \frac{C_{LO,i}}{P_{tol,i}} \tag{6.4}$$

Finally, looking at the formulas above it is possible to notice that, by adjusting at runtime the task periods within specific tolerance ranges, the utility optimisation leads to tune the overall system workload according to the specific computing resources available.

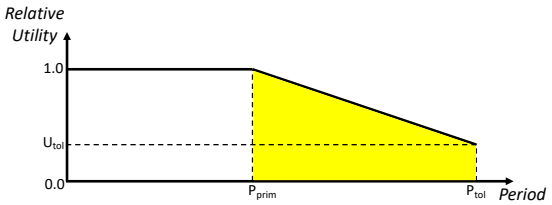
6.1.2 System Adaptation

The exploitation of the tolerance range described in sub-section 6.1.1 permits to optimise the load on each core by adjusting the tasks' periods. Each task has its own tolerance range $[P_{prim}, \ldots, P_{tol}]$ at which corresponds a utility range $[1, \ldots, U_{tol}]$. The runtime adaptation capability of a task is classified according to the relationship between its tolerance range and its tolerance utility as in Figure 6.2. The higher is the tolerance utility U_{tol} corresponding to P_{tol} and the larger is the tolerance range extent, the more it is possible to adjust the arrival rate of a task by preserving a high value of utility. Therefore, the adaptation at runtime is made considering first tasks that have a higher utility corresponding to the P_{tol} value and a larger tolerance range extent.

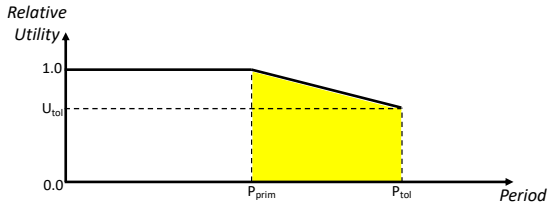
The system designer can set tolerance and utility ranges, defined as utility parameters up_i in Section 6.1.1, of each task according to specific application needs and periods larger than $P_{tol,i}$ are considered not beneficial anymore to guarantee an acceptable level of service. The rationale underlying the utility-based adaptation is that to guarantee the maximum possible achievable utility related to the possibility to tune the arrival rate among tasks that have to be guaranteed at same level of assurance. Therefore, among tasks with same criticality, tasks are deallocated from a core according to their capability adaptation, i.e., first are dropped tasks corresponding to Figure 6.2.d), then tasks in Figure 6.2.c), next tasks in Figure 6.2.b) and lastly tasks in Figure 6.2.a).



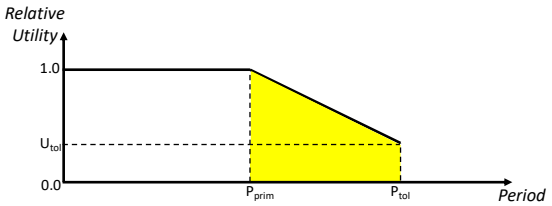
(a) Large tolerance range with high tolerance utility



(b) Large tolerance range with low tolerance utility



(c) Small tolerance range with high tolerance utility



(d) Small tolerance range with low tolerance utility

Figure 6.2: Service utility adaptation: tolerance range versus tolerance utility

6.2 Optimisation Method

The *Adaptive Tolerance-based Mixed-criticality Protocol* (ATMP) allows to tune the overall system utility by appropriately partitioning and adjusting tasks as soon as the number of cores working in a platform changes. Firstly, tasks are partitioned to cores prioritising higher criticality tasks. Secondly, on each core, the optimal adjustment of tasks is made by considering their tolerance interval, i.e., first tasks having least criticality and least benefit for utility optimisation are deallocated and then the arrival rates of the remaining tasks are optimised. The final overall system utility is computed by considering only the optimised

tasks kept allocated on each core that are deemed to be schedulable. As soon as more computing resources become available, tasks that were removed can be reallocated again, a further suitable periods optimisation is found on each core and this results in an increase of the overall system utility. Therefore, the variation of tasks' periods within their tolerance interval and the number of tasks kept onboard on each core affect the variation of system utility and this depends on the change of computing resources available.

In the whole, ATMP consists of the following two main parts:

1. Tasks are first sorted according to decreasing criticality. Then, the partitioning of tasks to cores is made as in Algorithm 1, i.e., highest criticality tasks are repeatedly selected and assigned to the core with least load allocated until all tasks are assigned. Note that Algorithm 1 is a static task partitioning scheme but with the special property that the tasks' criticality is taken into account for the allocation.

2. If a task set allocated to a specific core is schedulable, then it is processed by the underlying scheduler otherwise a binary search heuristics with linear programming optimisation is performed on each core as showed in Algorithm 2.

6.2.1 Criticality Aware Allocation

This sub-section describes the preliminary reallocation of tasks to cores made at runtime after the sudden fault of some processing element. Below is a description of the subroutines, either functions or procedures, used in Algorithm 1:

getTaskWithMaxCrit(τ)**:** it gets in input a list of tasks sorted by decreasing criticality and returns the first task with highest criticality in the list.

getCoreWithMinLoad(CS)**:** it gets in input the list of core ids in the system and returns the id of core c_{id} with least load allocated.

addTaskToCore(τ_i, c_{id})**:** it allocates the first task with highest criticality τ_{id} to the core c_{id} with least load allocated.

Algorithm 1 performs an online repartitioning of tasks to core after some core fails. Algorithm 1 considers a list of tasks sorted by decreasing criticality regardless of their priority or of their utilisation factor. At each iteration (line 2-6), the algorithm removes the first task from the queue, finds the core with least load allocated and then it allocates the task to such core.

Algorithm 1: Criticality aware allocation

Input : τ: list of tasks sorted by criticality;
CS: list of cores;

```
1 begin
2   while τ is not empty do
3     τ_i ← getTaskWithMaxCrit(τ);
4     c_id ← getCoreWithMinLoad(CS);
5     addTaskToCore(τ_i, c_id);
6   end
7 end
```

6.2.2 The ATMP Utility Optimisation

This sub-section describes the utility optimisation made on a set of tasks allocated to one core. A precise description of how the heuristics works is made in Algorithm 2. Below is an explanation of subroutines used in the pseudocode:

isSchedulable(τ)**:** it returns `true` if τ is deemed to be schedulable or `false` otherwise. This schedulability test checks the three assumptions introduced in Chapter 4 since we are assuming task set with two WCET estimates that are schedulable by a mixed-criticality scheduling protocol.

$\text{load}_{\text{tol}}(tts)$**:** it represents the utilisation factor of the task set tts computed by using the optimistic WCET estimate $C_{LO,i}$ and the tolerance period $P_{tol,i}$ for each task $\tau_i \in tts$.

removeMinCritTask(tts)**:** it removes the first task with worst capability adaptation (as described in Section 6.1.2 and Figure 6.2) among those with least criticality.

solveILP(tts)**:** it generates a linear programming problem as described in [6] and returns as result a list of optimised periods.

applyPrimaryPeriods($tts, periods$)**:** it replaces the optimised periods in the task set tts.

applyPrimaryPeriod(tts)**:** it sets the primary period as optimal period when tts consists of just one task, no optimisation is performed.

average($lm, lm2$)**:** it computes the middle value between two interval ends and then returns it.

Once a set of tasks has been assigned to a specific core, Algorithm 2 first checks its schedulability as indicated in line 2. If the task set is deemded to be schedulable according to the specific test used, then it will be processed with the underlying scheduling protocol. Conversely, if the task set is deemed to be not schedulable, then a binary search is performed for a predefined number of times indicated by the variable $lcnt$ (line 5-28). At every iteration, a copy of the partitioned task set tts assigned to such core is modified. Every time, the binary search finds an lm value, initially set to a default value, to use as load upper bounds for the tolerance load of task set tts. While the utilisation factor of the set of tasks tts computed according to the tolerance periods is greater than the upper bounds lm, then tasks with worst adaptation capability (Figure 6.2) among those with least criticality are dropped (line 7-9).

Once a set of tasks with tolerance load suitable with the upper bounds lm is found, the heuristics finds the optimised arrival rates by means of the LP problem described in [6] and then replaces such periods within each task on the core (line 10-15). Note that if the task set with tolerance load not greater than lm consists of just one task (line 14), then it is assumed to have a load less than or at maximum equal to 100% and, since there is no interference from higher priority tasks, it will be schedulable by default. In this particular case, no optimisation is performed and the task is stored in the tts variable.

A schedulability test chosen according to the underlying scheduling protocol checks if the set of tasks with optimised periods will complete within their deadlines (indicated at line 16). If the optimised task set is deemed to be schedulable and its load is greater than that of the last feasible optimised task set, then it is stored in the τ_c^* variable and the range in which performing the binary search is updated to continue in the upper half of the interval (line 16-22). Otherwise, if the optimised task set is not schedulable, then the range in which performing the binary search is updated to continue in the lower half of the interval (line 23-26). At the end of each iteration the counter $lcnt$ is decreased (line 27).

The algorithm ends when $lcnt = 0$ and it returns the optimised set of tasks assigned to a specific core with the best tolerance load (line 30).

Although Algorithm 2 can be used with any schedulability test, we use the three step schedulability analysis introduced in [69] leading to process task sets respecting the three assumptions used in Chapter 4. Therefore, as specified in the system model section 6.1, each task has got one or two WCET estimates, indicating respectively a low and a high level of assurance for its correct completion.

Algorithm 2: ATMP Utility Optimisation

Input : τ_c: task set allocated to a core;

Local : $lm \leftarrow 0.90$;
$lm1 \leftarrow 0.2$;
$lm2 \leftarrow 1.0$;
$lcnt \leftarrow 6$;
$tts \leftarrow Null$;
$bestlm \leftarrow 0.0$;

Output: τ_c^*: optimised task set;

```
1  begin
2      if isSchedulable(τ) then
3          τ_c^* ← τ_c;
4      else
5          while lcnt > 0 do
6              tts ← τ;
7              while (load_tol(tts) > lm) do
8                  removeMinCritTask(tts);
9              end
10             if len(tts) > 1 then
11                 periods ← solveILP(tts);
12                 tts ← applyPrimaryPeriods(tts, periods);
13             else
14                 tts ← applyPrimaryPeriod(tts);
15             end
16             if isSchedulable(tts) then
17                 if lm > bestlm then
18                     bestlm ← lm;
19                     τ_c^* ← tts;
20                 end
21                 lm1 ← lm;
22                 lm ← average(lm, lm2);
23             else
24                 lm2 ← lm;
25                 lm ← average(lm1, lm);
26             end
27             lcnt ← lcnt − 1;
28         end
29     end
30     return τ_c^*;
31 end
```

6.3 Safety Implications of Scheduling

Safety-critical systems are typically subject to two related but different correctness requirements that is necessary to consider during their design phase: a priori verification and run-time robustness [12]. The verification determines offline whether a system will behave correctly during runtime and deals with the case when runtime behaviour is compliant with its assumed model while the robustness at runtime is concerned with what happens when modelling assumptions are violated.

ATMP manages these cases in which the a-priori assumptions are violated and ensures that performances degrade gracefully whenever computing resources suddenly become insufficient. In such cases, a general rule is that less important system functionalities should be compromised before the most important ones. In fact, each task within a task system can be classified on the extent on which it contributes to the safety of the system. As an example, in case of just two criticalities, tasks can be classified into safety-related and non-safety related, corresponding to HI and LO in the system model used in Chapter 4. A typical subdivision of tasks, even in case of multiple criticality levels, may be based on *Safety Integrity Levels* (SIL) like in IEC61508 standard [108] or similar classifications. Traditionally, the safety-related and non-safety related functions are required to be separated to not interfere among them [108], since failures of non-safety related functions should not cause a dangerous failure of the safety functions. This issue is even more a challenge nowadays with the increasing trend in designing systems having tasks of different criticality running on a shared platform [12].

The ATMP protocol allows to keep the advantages of the different WCET estimation process per different criticality levels that enables to design systems in which the safety-critical tasks execution is guaranteed to a higher level of assurance while making a more efficient resource usage at runtime since for non critical tasks lower guarantees are required. From this point of view, the analysis introduced by Vestal et al. permits to verify the correctness of task systems that otherwise would have been deemed unschedulable using conventional analysis techniques.

Furthermore, ATMP allows to drop lower criticality tasks with no impact on the performances of the higher criticality ones. In fact, if any core suddenly becomes unavailable, ATMP allows to de-allocate tasks according to their criticality and online adaptation capability till when the computing resources become available again. Whenever some processing elements that failed at some point

becomes active again, then ATMP can re-deploy tasks that were de-allocated. However, discarding non-safety related tasks for long time is not free from consequences since they can be important for non-safety reasons. The tolerance range present in each task allows to degrade also the service provided to higher criticality tasks to keep running as many non-critical tasks as possible.

Safety-criticality is of particular concern for the UK critical national infrastructures supplying essential services. These services include provision of drinking water, transport of oil and gas, rail transport and medical infrastructure. Moreover, the EU has recently established a directive [109] that mandates that providers take steps to mitigate the impact of incidents which can compromise the delivery of essential services, and the British National Cyber Security Centre guidance [110] identifies as core principle that such essential services must be resilient, meaning that their provision should not be interrupted. In some cases the infrastructure for the provision of essential services may also provide additional services classified as non-essential (e.g., a system monitoring that provides functionality for availability of drinking water may also be used to monitor the supply of non-potable water). Failure of essential services typically has safety implications, while temporary failure of non-essential services is unlikely to represent a safety risk, intended as a potential risk for human lives. Because of this, tasks associated with the essential service are regarded as of higher criticality and those associated only with the non-essential services as lower criticality ones. In case of resource shortages, the ATMP protocol removes first the non-essential services with worst adaptation capability and meanwhile it ensures the continuity of essential services during and after the resolution of the incident. Furthermore, ATMP guarantees a timely restoring of non essential services whenever the faulty computing elements within the infrastructure are reactivated. As such, ATMP represents a potential solution for scheduling tasks on a shared platform required to comply with the European and British directives on network and information systems security [109, 110].

6.4 Chapter Summary

This chapter introduced the ATMP protocol, an adaptive and criticality-aware heuristics that optimises the overall system utility by adjusting the system workload of individual tasks. ATMP allows to allocate and de-allocate tasks from cores at run-time whenever processing elements fail or become available again during the system lifetime.

Section 6.1 introduced the system and task model that uses the utility functions as designing instrument for system adaptation at runtime. The presence of the tolerance interval allows to appropriately optimise each task arrival rate. The system adaptation is made by respecting each predefined task utility requirements. Section 6.2 describes the ATMP heuristics used for system optimisation. The task removal is made according to the possibility to profitably adjust the arrival rates at runtime in order to keep as many tasks running as possible. Lastly, in Section 6.3 I explained how ATMP can address the safety issues emerged from the European and British guidances for network and system security [109, 110].

Chapter 7

Experimental Evaluation

This chapter describes the experiments conducted first with LBP and its variants and then with ATMP.

Section 7.1 describes the experiments made with the protocols for short-term scheduling on single-core architectures while Section 7.2 contains the experimental evaluation about the mid-term re-allocation and optimisation strategy on multi-cores systems. All the results are collected within tables and then analysed by means of figures and charts. Section 7.3 concludes and summarises the whole chapter.

7.1 Evaluation of Lazy Bailout Protocols

This section describes the experiments made with the fixed-priority scheduling protocols designed for uniprocessor platforms. It contains the metrics used to evaluate the amount of jobs scheduled, the configuration settings and scenarios in which such experiments have been made and lastly the final outcome. In particular, sub-Section 7.1.3 compares the lazy bailout approaches with the state of the art fixed-priority scheduling protocols in hard real-time settings, i.e., jobs not completed within their deadlines are abandoned. Then, since LBP and SLBP have different scheduling behaviour when tasks have deadlines smaller than their periods, in sub-Section 7.1.4 I also compared LBP with its variant SLBP in soft real-time settings by considering jobs that complete after their deadlines. All the experiments also consider the scheduling protocols derived from the integration of the basic methods with complementary techniques like the offline sensitivity analysis and the gain time collection at runtime.

7.1.1 Setup of Experiments

This sub-section contains the explanation of software and settings used to conduct the experiments. I have developed a scheduling framework in Python 2.7.10 that first creates task sets that are schedulable according to AMC-rtb [69] and then runs sequentially different scheduling methods to process them. A final report is created to summarise, compare and analyse the results.

I have conducted different experiments first with implicit and then with constrained deadline task sets. The task model used is that introduced in Chapter 4 that considers dual-criticality task sets with independent periodic tasks. HI tasks have two WCETs, one more conservative indicated with C_{HI} and another more optimistic indicated with C_{LO}, while for LO tasks only the knowledge of unsafe upper bounds C_{LO} is assumed. Each experiment consists of a group of three thousand task sets randomly generated. The number of tasks within each task set varies randomly between four and twelve. Within each task set, the amount of HI tasks varies between the 20% and 70%. Priorities are assigned to tasks according to *Deadline Monotonic* (DM) strategy in which task instances with the shortest deadline have the highest priority. Three task set scenarios have been created by appropriately combining deadlines of each task within a task set. Every task set group created to make experiments belongs to one of the three scenarios specified below.

HC-LP: The first case contains job sets where all HI jobs have larger deadlines than all LO jobs. Therefore, all HI jobs have lower priority than all the LO jobs:

$$\forall j \in J_{HI} \wedge \forall j' \in J_{LO}.\ pr(j) < pr(j')$$

HC-MP: This case contains job sets where HI jobs could have deadlines that are either smaller or larger than those of LO ones. Therefore, HI and LO jobs have mixed priorities:

$$\forall j \in J_{HI} \wedge \forall j' \in J_{LO}.\ pr(j) \leq pr(j') \vee pr(j) > pr(j')$$

HC-HP: It contains job sets where all HI jobs have smaller deadlines than all LO ones. This implies that all HI jobs have higher priority than LO jobs:

$$\forall j \in J_{HI} \wedge \forall j' \in J_{LO}.\ pr(j) > pr(j')$$

It is important to notice that, if higher criticality tasks have all higher priority than lower criticality ones, then the scheduling problem so created becomes equivalent to the standard real-time scheduling problem since there is no criticality inversion. The same applies to those cases in which higher priority is assigned to the highest criticality tasks regardless of their periods or deadline as in *Criticality As Priority Assignment* (CAPA) [13].

7.1.2 Performance Metrics and Evaluation Scenarios

This sub-section introduces the criteria used to assess performances of mixed-criticality scheduling protocols.

To evaluate the results, I defined two types of metrics. The former is relative to the whole amount of task sets while the latter is relative to jobs within each individual task set. I named them respectively task set and global job set schedulability.

The task set schedulability formula *tsched* is defined as follows:

$$tsched(S, cat) = \frac{|STSsucc(S, cat)|}{|S|} \tag{7.1}$$

where S could be either a simple task set τ or set of task sets STS and the category $cat \in \{HI + LO, HI, LO\}$ represents the type of tasks within a set that is HI for high-criticality tasks, LO for low-criticality tasks and either in case of HI+LO usage. The function *STSsucc* depends on the scheduling protocol that is actually used and returns as output the set of task sets STS in which there are no jobs missed of category *cat*. The absolute values within the formula give the set cardinality. The equation 7.1 allows to derive the percentages of tasks set in STS that are successfully processed according to the category *cat* as follows:

TSSched: amount of task sets scheduled with no jobs missing their deadlines.

$$TSSched = tsched(STS, HI + LO)$$

TSSchedHI: amount of task sets scheduled with no HI jobs missing their deadlines.

$$TSSchedHI = tsched(STS, HI)$$

TSSchedLO: amount of task sets scheduled with no LO jobs missing their deadlines.

$$TSSchedLO = tsched(STS, LO)$$

The task set schedulability permits to show the percentage of task sets in which no job of category *cat* misses its deadline. However, whenever a task set contains some jobs that miss their deadline, it is also useful to assess the level of service provided in terms of jobs completed and jobs abandoned or aborted. Such a view is provided by the amount of jobs that are completed within or even after their deadlines. The job set completion rate methods $jsched$ and $jsched^*$ are defined with this regard. In particular, $jsched$ returns only the percentage of jobs of category *cat* generated by a specific task set that complete within their deadlines while $jsched^*$ returns the total percentage of jobs of category *cat* generated by a task set that complete, either within or after their deadline.

The on-time job set completion rate $jsched$ is formally written as below:

$$jsched(\tau_{cat}) = \frac{|Jsucc(J(\tau_{cat}))|}{|J(\tau_{cat})|} \quad | \; cat \in \{LO, HI\} \tag{7.2}$$

while the total job set completion rate $jsched^*$ is defined as follows:

$$jsched^*(\tau_{cat}) = \frac{|Jsucc^*(J(\tau_{cat}))|}{|J(\tau_{cat})|} \quad | \; cat \in \{LO, HI\} \tag{7.3}$$

The formulas above are used to compute the global job set on-time and total completion rates showed below that return respectively the average amount of jobs of category *cat* completed within their deadline and the overall percentage of jobs completed, including those that terminate after their deadline. It is worth to notice that HI jobs all complete within their deadlines, thus only the global job set on-time completion average is computed for them. The global job set completion rates are computed on the whole amount of task sets STS and are shown below:

$$gjsched(STS, cat) = \frac{\sum_{\tau \in STS} jsched(\tau_{cat})}{|STS|} \quad | \; cat \in \{LO, HI\} \tag{7.4}$$

$$gjsched^*(STS, cat) = \frac{\sum_{\tau \in STS} jsched^*(\tau_{cat})}{|STS|} \quad | \; cat \in \{LO, HI\} \tag{7.5}$$

As in the previous case, it is possible to filter the jobs completed according to the category *cat* as below:

GJSched: average number of jobs (either HI or LO) generated by a set of task

set that complete within their deadlines.

$$GJSched = gjsched(STS, HI + LO)$$

*GJSched**: average number of jobs (either HI or LO) generated by set of task set that complete, including those that terminte after their deadlines.

$$GJSched = gjsched(STS, HI + LO)$$

GJSchedHI: average number of HI jobs generated by a set of task set that complete within their deadlines.

$$GJSchedHI = gjsched(STS, HI)$$

GJSchedLO: average number of LO jobs generated by a set of task set that complete within their deadlines.

$$GJSchedLO = gjsched(STS, LO)$$

*GJSchedLO**: average number of LO jobs generated by a set of task set that complete, including those that complete after their deadlines.

$$GJSchedLO^* = gjsched^*(STS, LO)$$

It is possible to use the performance metrics described above to assess the scheduling methods in different settings and analyse results in each case.

7.1.3 Discussion of Results in Hard Real-Time Settings

This sub-section describes the assessment conducted by comparing the LBP protocols with the former BP approaches. I have conducted different experiments, each consisting of a group of three thousand implicit deadline task sets randomly generated. The following scheduling protocols have been compared:

- the standard *Fixed-Priority Preemptive Scheduling* with DM as priority assignment (FPPS-DM).
- the standard *Bailout Protocol* (BP).
- the *Lazy Bailout Protocol* (LBP).

- the *Soft Lazy Bailout Protocol* (SLBP).
- the *Bailout Protocol - Slack* (BPS), *Lazy Bailout Protocol - Slack* (LBPS) and the *Soft Lazy Bailout Protocol - Slack* (SLBPS) deriving from the integration of the basic mixed-criticality protocols with the offline sensitivity analysis [106, 44] while guaranteeing the schedulability according to AMC-rtb [69].
- the *Bailout Protocol with Gain time* (BPG), *Lazy Bailout Protocol with Gain time* (LBPG), and *Soft Lazy Bailout Protocol with Gain time* (SLBPG) where each job that finishes before its optimistic time threshold in Normal mode gives its gain time to increase the time budget of next job ready to be scheduled.
- the *Bailout Protocol - Slack and Gain time* (BPSG), the *Lazy Bailout Protocol - Slack and Gain time* (LBPSG) and the *Soft Lazy Bailout Protocol - Slack and Gain time* (SLBPSG) deriving from the integration of both the offline scaling of C_{LO} of HI tasks with sensitivity analysis and the online gain time collection with the basic scheduling protocols.

Tables 7.1 and 7.2 contain respectively the results about the task set schedulability and global job set completion rates of mixed-criticality scheduling methods. Since these experiments are made with set of tasks having deadline equal to periods, all LO jobs deadlines are considered to be hard. I have also collected data within figures to summarise the results of experiments with dual-criticality task sets and show the results in all the three different scenarios. More precisely, Figure 7.1, Figure 7.2 and Figure 7.3 show the averages of task sets and jobs scheduled while Figure 7.4, Figure 7.5 and Figure 7.6 show how the LO jobs scheduled are distributed.

Figure 7.1 and Figure 7.2 summarise the results in cases where there is criticality inversion. In these situations, if no HI job completes within its optimistic threshold estimate C_{LO}, then very likely there will be some new incoming higher priority LO jobs that will interfere with it. Conversely, Figure 7.3 contains information about cases in which all HI jobs have higher priority than LO jobs, i.e., all the critical jobs have smaller deadlines. This basically leads to have no interference between HI and LO jobs and thus no criticality inversion occurrence during the scheduling process.

Looking both at task set and job set schedulabilities results, it is possible to notice that the standard deadline monotonic approach always schedules jobs only according to priorities. In this case, the percentages of HI or LO jobs successfully

scheduled mainly depend only on their priority, with all LO jobs that meet their deadlines in HC-LP scenario, i.e., LO jobs have smaller deadlines than HI jobs, and all HI jobs that always meet their deadlines in HC-HP scenario, i.e., HI jobs have smaller deadlines than LO jobs.

On the other hand, the mixed-criticality protocols always assure that there are no HI jobs missed regardless of job priorities. The experiments confirm what is stated in Chapter 5 with LBP that always successfully schedules more LO jobs than BP since BP schedules no more than 7.07% of task sets with no jobs missed while LBP can schedule till the 52.33% of task sets with no jobs missed. All figures highlight that the amount of jobs scheduled further increases when the offline and online complementary techniques are used. It is worth to notice that the usage of sensitivity analysis and the gain time mechanism always leads to have the same effects when applied both to the standard or to the lazy bailout methods. A noticeable result is that each LBP-based approach allows to complete more LO jobs within their deadlines than the corresponding standard BP-based protocol. In the whole, according to the criteria defined in Chapter 5, LBPSG and SLBPSG are the protocols that increase more the amount of jobs completed within their deadlines. As an example, LBPSG and SLBPSG schedule between the 43.07% and 58.67% of task sets with no jobs missed compared to BPSG for which the percentage of set of tasks with no jobs missed is at maximum 26.87%.

As a conclusion, LBP and SLBP always schedule more LO jobs compared with BP while guaranteeing the same level of performances in processing HI jobs. Each protocol can be further refined by exploiting the system slack time identified offline and the online gain time collection to still increase the amount of lower criticality jobs scheduled. With regard to the formal evaluation criteria introduced in Chapter 5, the results show that LBPS and SLBPS always outperform BPS, LBPG and SLBPG always outperform BPG and LBSG and SLBPSG always outperform BPSG. Finally, the usage of mixed-criticality protocols is recommended in HP-LP and HC-MP scenarios, i.e., when HI jobs could have lower priorities than LO jobs.

Figure 7.4, Figure 7.5 and Figure 7.6 display the distribution of the LO jobs percentages per task set that are completed within their deadlines. Each scheduling protocol is represented by a box-and-wisker diagram with the box itself representing the range in which at least the 50% of results tend to be concentrated. The box also contains the indication of the median and the mathematical average of all the LO jobs scheduled by the related protocol. The results highlight how the LBP/SLBP-based methods always increase the LO jobs success rate, as defined in Chapter 5, compared with the former BP ones.

	HC-LP			***HC-MP***			***HC-HP***		
Method	*TSSched*	*TSSchedHI*	*TSSchedLO*	*TSSched*	*TSSchedHI*	*TSSchedLO*	*TSSched*	*TSSchedHI*	*TSSchedLO*
FPPS-DM	83.03	83.03	100.0	66.13	97.20	66.60	68.80	100.0	68.80
BP	7.07	100.0	7.07	2.63	100.0	2.63	4.40	100.0	4.40
BPG	11.90	100.0	11.90	3.10	100.0	3.10	4.53	100.0	4.53
BPS	17.07	100.0	17.07	16.80	100.0	16.80	21.17	100.0	21.17
BPSG	24.17	100.0	24.17	22.13	100.0	22.13	26.87	100.0	26.87
LBP	27.97	100.0	27.97	32.83	100.0	32.83	52.33	100.0	52.33
LBPG	34.23	100.0	34.23	33.53	100.0	33.53	52.70	100.0	52.70
LBPS	36.27	100.0	36.27	40.13	100.0	40.13	55.53	100.0	55.53
LBPSG	43.07	100.0	43.07	44.07	100.0	44.07	58.67	100.0	58.67
SLBP	27.97	100.0	27.97	32.83	100.0	32.83	52.33	100.0	52.33
SLBPG	34.23	100.0	34.23	33.53	100.0	33.53	52.70	100.0	52.70
SLBPS	36.27	100.0	36.27	40.13	100.0	40.13	55.53	100.0	55.53
SLBPSG	43.07	100.0	43.07	44.07	100.0	44.07	58.67	100.0	58.67

Table 7.1: BP and LBP variants: comparison of task set schedulability (%)

7.1.4 Comparison of LBP with SLBP in Soft Real-Time Settings

This sub-section describes the outcome of the experiments conducted by comparing LBP and SLBP based scheduling protocols. The evaluation considers both jobs that complete within their deadlines as well as jobs that complete after their deadlines. The results show that the SLBP always increases the amount of jobs completed after their deadlines but sometime at the expense of those scheduled within their deadlines.

The experiment considers a group of three thousand task sets randomly generated. Each task set consists of tasks with constrained deadlines, with deadline that can be even 50% smaller than its period. Since deadlines of tasks are randomly generated and mostly not coincide with their related periods, the outcome only contains the case in which HI and LO jobs in each task set have mixed priorities, formally defined as HC-MP in sub-section 7.1.2. Because of the schedulability assumptions, all HI jobs complete within their deadlines. As in sub-section 7.1.3, the number of HI tasks per task set varies randomly between the 20% and 70%.

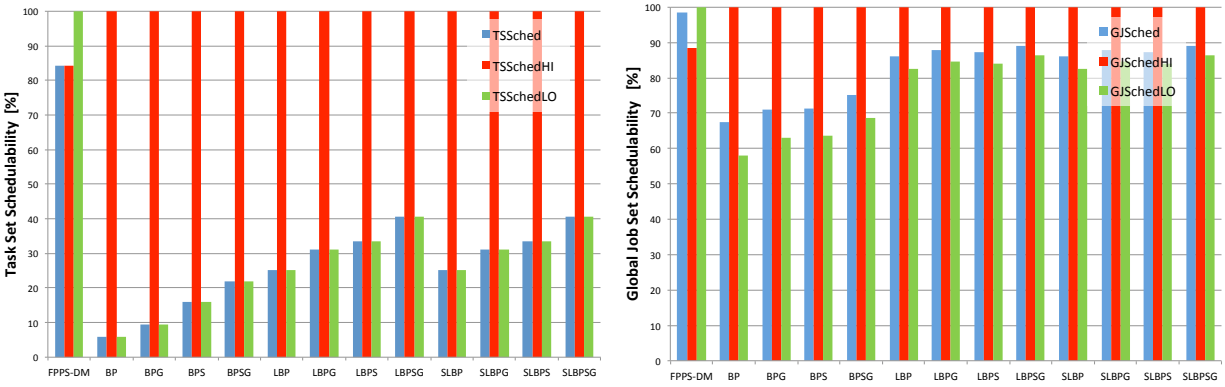
	HC-LP			***HC-MP***			***HC-HP***		
Method	*GJSched*	*GJSchedHI*	*GJSchedLO*	*GJSched*	*GJSchedHI*	*GJSchedLO*	*GJSched*	*GJSchedHI*	*GJSchedLO*
FPPS-DM	98.32	86.94	100.0	96.25	99.10	94.49	96.74	100.0	93.56
BP	67.71	100.0	59.32	72.46	100.0	55.22	82.05	100.0	58.91
BPG	71.21	100.0	64.07	73.19	100.0	56.26	82.32	100.0	59.43
BPS	71.46	100.0	64.39	77.56	100.0	63.87	85.46	100.0	67.56
BPSG	74.99	100.0	69.18	79.79	100.0	67.64	86.95	100.0	71.34
LBP	85.68	100.0	82.35	90.29	100.0	85.18	94.72	100.0	89.14
LBPG	87.55	100.0	84.67	90.46	100.0	85.40	94.74	100.0	89.18
LBPS	86.77	100.0	83.87	90.72	100.0	85.81	94.80	100.0	89.32
LBPSG	88.63	100.0	86.18	91.20	100.0	86.59	94.93	100.0	89.67
SLBP	85.68	100.0	82.35	90.29	100.0	85.18	94.72	100.0	89.14
SLBPG	87.55	100.0	84.67	90.46	100.0	85.40	94.74	100.0	89.18
SLBPS	86.77	100.0	83.87	90.72	100.0	85.81	94.80	100.0	89.32
SLBPSG	88.63	100.0	86.18	91.20	100.0	86.59	94.93	100.0	89.67

Table 7.2: BP and LBP variants: comparison of jobs scheduled within their deadline (%)

As stated in Chapter 4, the LBP protocol and, as a consequence, its derivatives (LBPG, LBPS and LBPSG) treat LO jobs as soft real-time jobs *only* during the high criticality execution, i.e., when the system runs in Bailout or Recovery modes. On the other hand, SLBP and its derivatives allow to LO jobs to complete after their deadlines even during the Normal mode.

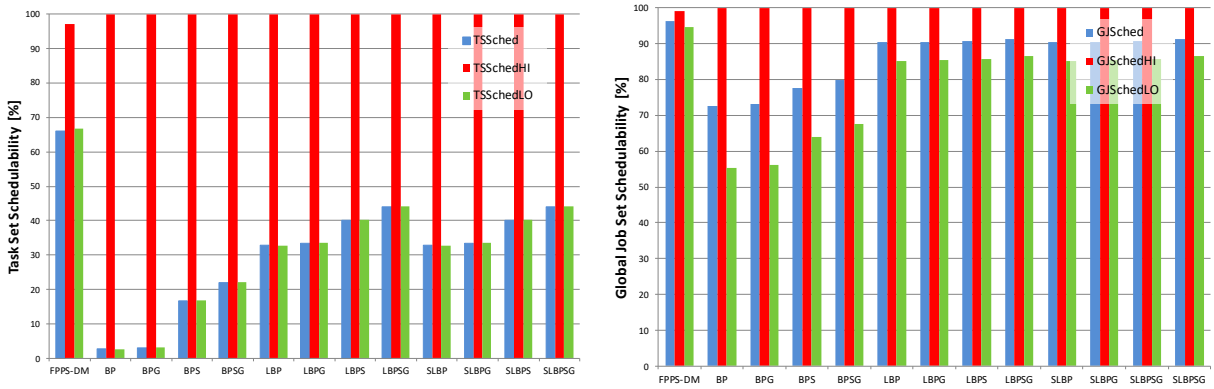
Table 7.3 and Table 7.4 show respectively data regarding the task schedulability and the global on-time and total jobs completion averages. The average completion rate including tardy jobs is not shown for HI jobs since they always complete within their deadlines.

Table 7.3 shows that LBP and SLBP process the same amount of task sets with no jobs missing their deadlines. However, whenever task sets are not feasible, Table 7.4 reveals that there are some cases in which LBP schedules more LO jobs within their deadline while SLBP always increases the overall amount of LO jobs completed, including tardy jobs. This is due to the different usage of system idle time made by the two protocols in scheduling LO jobs. LBP removes jobs from the low-priority queue as soon as they miss their deadlines and



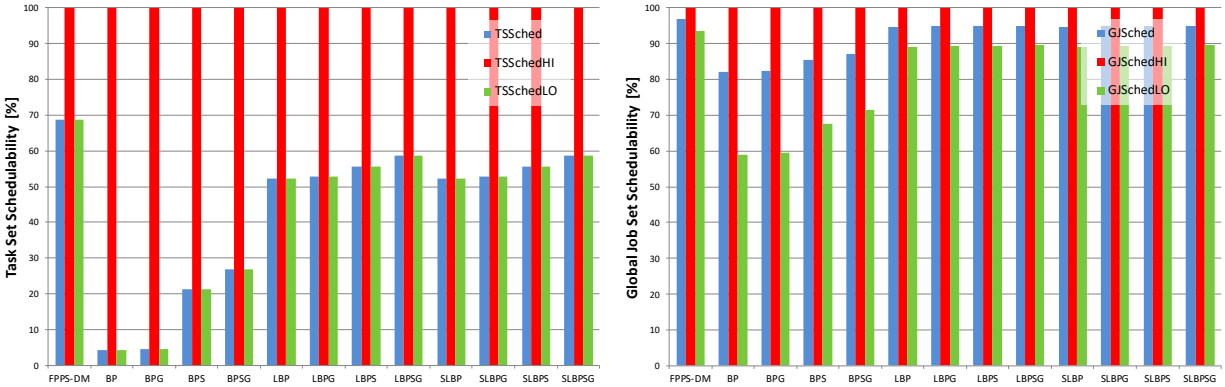
(a) Task sets with no jobs missed (b) Average of jobs scheduled per task set

Figure 7.1: BP and LBP variants: schedulability in HC-LP scenario



(a) Task sets with no jobs missed (b) Average of jobs scheduled per task set

Figure 7.2: BP and LBP variants: schedulability in HC-MP scenario



(a) Task sets with no jobs missed (b) Average of jobs scheduled per task set

Figure 7.3: BP and LBP variants: schedulability in HC-HP scenario (as priority and criticality values have the same order, this is essentially a standard real-time scheduling problem)

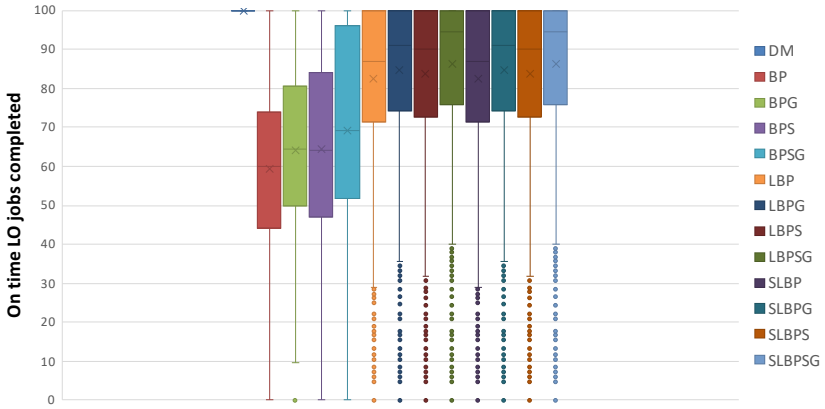


Figure 7.4: BP and LBP variants: LO jobs scheduled per task set in HC-LP scenario

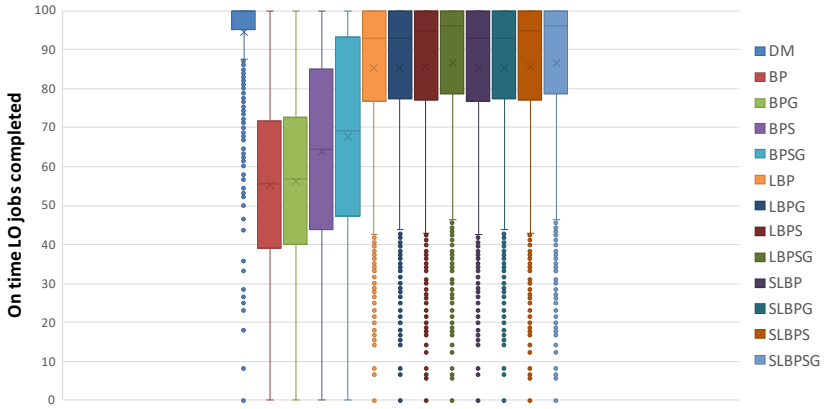


Figure 7.5: BP and LBP variants: LO jobs scheduled per task set in HC-MP scenario

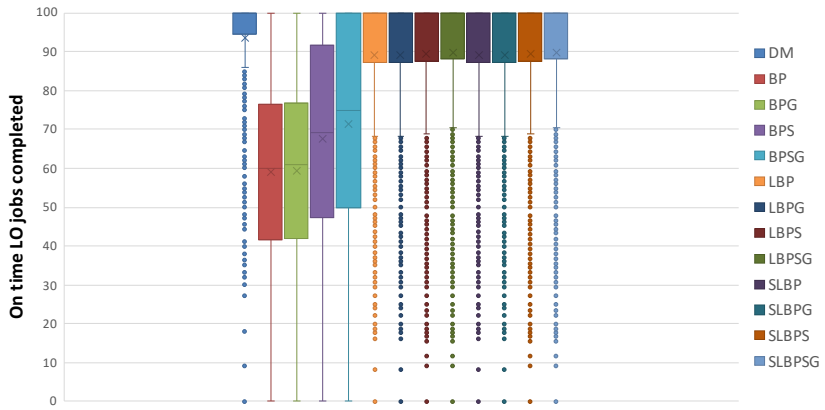


Figure 7.6: BP and LBP variants: LO jobs scheduled per task set in HC-HP scenario

Method	*TSSched*	*TSSchedHI*	*TSSchedLO*
LBP	31.47	100.0	31.47
LBPG	31.93	100.0	31.93
LBPS	35.67	100.0	35.67
LBPSG	38.67	100.0	38.67
SLBP	31.47	100.0	31.47
SLBPG	31.93	100.0	31.93
SLBPS	35.67	100.0	35.67
SLBPSG	38.67	100.0	38.67

Table 7.3: LBP and SLBP derivatives: comparison of task set schedulability (%)

Method	*GJSched*	*GJSched**	*GJSchedHI*	*GJSchedLO*	*GJSchedLO**
LBP	89.96	90.77	100.0	82.23	83.79
LBPG	90.02	90.84	100.0	82.33	83.91
LBPS	90.21	91.06	100.0	82.67	84.32
LBPSG	90.47	91.33	100.0	83.18	84.83
SLBP	89.81	92.01	100.0	82.00	86.04
SLBPG	89.88	92.08	100.0	82.09	86.14
SLBPS	90.07	92.19	100.0	82.44	86.38
SLBPSG	90.34	92.40	100.0	82.95	86.76

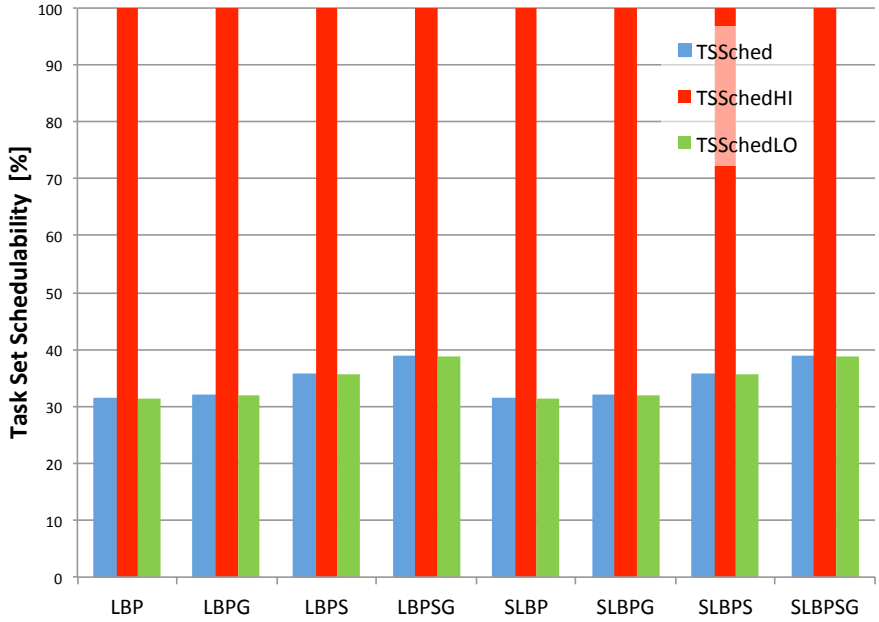
Table 7.4: LBP and SLBP derivatives: average of completion rates of jobs (%)

by doing this it makes room for new incoming LO jobs that might successfully complete. On the other hand, SLBP allows to LO jobs to complete even after their deadlines which takes up extra resources compared to LBP.

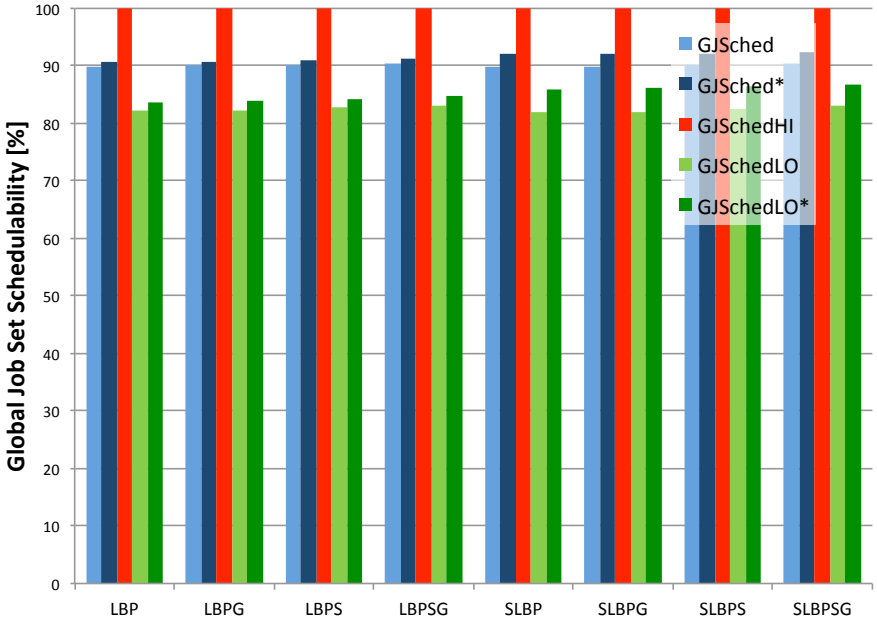
Figure 7.7.a) and Figure 7.7.b) represent graphically what is shown respectively in Table 7.3 and Table 7.4. On the other hand, Figure 7.8.a) and Figure 7.8.b) contain the distribution of LO jobs scheduled within and after their deadlines.

7.2 Experimental Evaluation of ATMP

This section assesses and analyses the ATMP protocol, the experiments target task sets running on multi-core platforms in which some processing elements can fail at run-time and it is necessary to remap tasks to cores. The analysis shows that the approach based on utility accrual and tolerance range exploitation out-



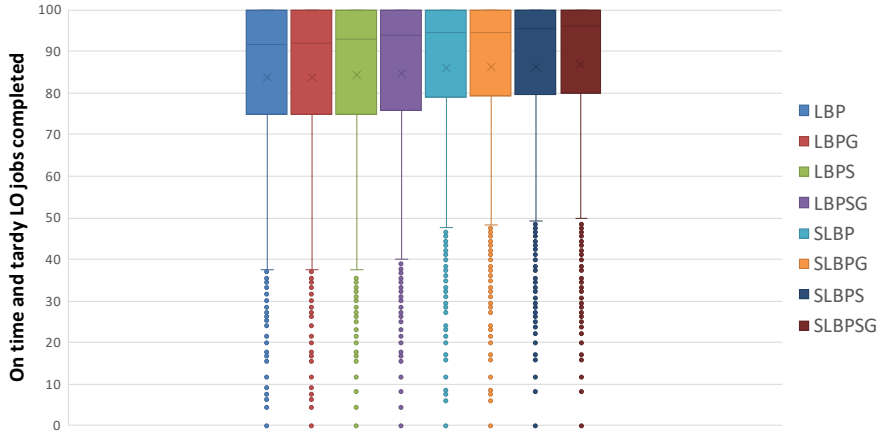
(a) Task sets with no jobs missed



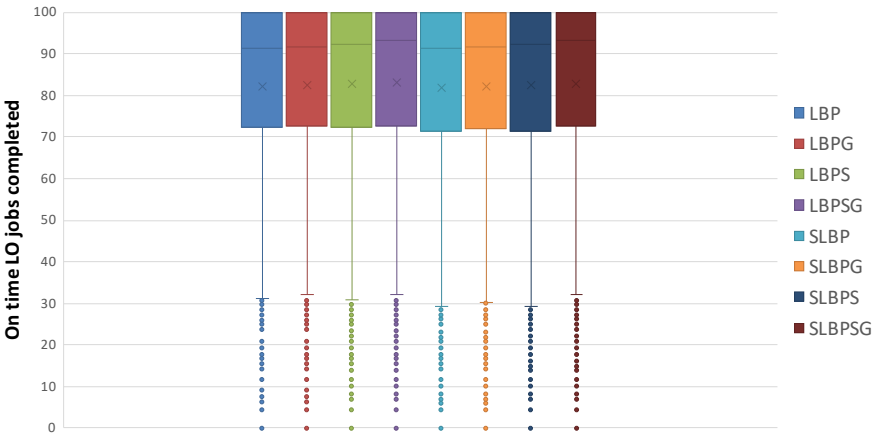
(b) Jobs average completion rates

Figure 7.7: LBP and SLBP derivatives: comparison between LBP and SLBP in soft real-time settings

performs the classical reallocation of tasks to cores based just on the knowledge of criticality. The optimisation based on the tolerance range exploitation permits to keep onboard more tasks and to increase the overall system utility accumulated.



(a) LO jobs completed, including tardy jobs



(b) LO jobs completed within their deadlines

Figure 7.8: LBP and SLBP derivatives: distribution of LO jobs completed

To show the effectiveness of ATMP, I compared it with a standard approach, referred to as *Standard Adaptive Mixed-criticality Protocol* (SAMP), in which mixed-criticality tasks have no tolerance range. With the SAMP approach, the tasks removal is performed only considering the load computed according to the predefined periods and no LP optimisation is made. On the other hand, ATMP adjusts the tasks' periods within a predefined tolerance range according to specific needs at run-time.

The task model used to create task sets is that contained in Chapter 6. I have created a task set consisting of twenty tasks randomly generated and then I have processed it using both ATMP and SAMP first on eight, then on five and finally on three cores. The experiment confirms that, in case of resource shortages, i.e.,

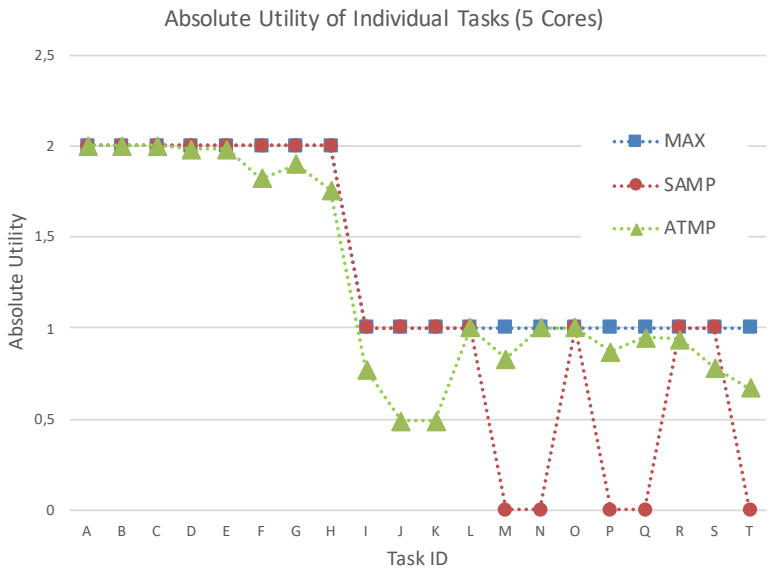
sudden unavailability of computing resources, the usage of the tolerance range to appropriately optimise the tasks arrival rates allows to ATMP to de-allocate a smaller amount of tasks per core. Both approaches worked well with eight cores since no task was removed. However, ATMP showed its advantages after further reducing the number of processing elements. Because of this, I only show the comparison of performances between SAMP and ATMP in case of five and three cores. Figure 7.9 displays the absolute utility accrued by each individual task with the two above approaches compared with the maximum achievable utility indicated with MAX. The absolute utility of de-allocated tasks is 0. Tasks from A to H have criticality 2 while tasks from I to T have criticality 1. Figure 7.9a) presents the runtime reallocation on five cores. In this case, SAMP removes five non-critical tasks while ATMP allows, via tolerance-based optimisation, to adjust the tasks periods and to keep all tasks allocated to their cores. It is worth to notice that such result is achieved also slowing down higher criticality tasks and this leads to a decrease in the overall load allocated on each core. Such results are even more emphasised in Figure 7.9b) in which the number of cores available is further reduced. In this latter case, in the whole, SAMP removes thirteen tasks (two of which are highly critical ones) while ATMP removes just six tasks and keeps onboard all the higher criticality tasks.

Table 7.5 summarises the overall outcome of the experiment by showing the total relative and absolute utilities accrued within the system and the amount of tasks removed respectively by SAMP and ATMP. The total relative utility consists of the sum of the individual task utility while the total absolute utility considers also the task criticality.

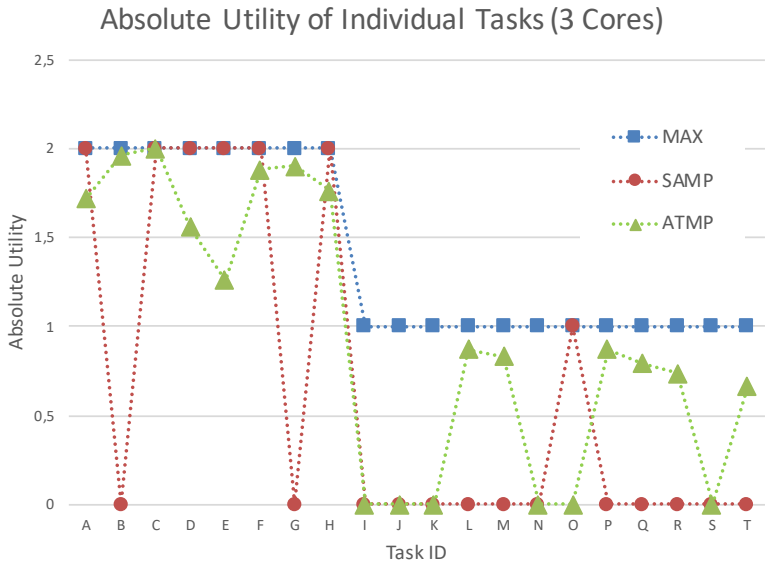
	SAMP			*ATMP*		
Cores#	*Rel. Utility*	*Abs. Utility*	*Task dropped*	*Rel. Utility*	*Abs. Utility*	*Tasks dropped*
8	20.00	28.00	0	20.00	28.00	0
5	15.00	23.00	5	17.51	25.23	0
3	7.00	13.00	13	11.79	18.81	6

Table 7.5: Overall comparison between *ATMP* and *SAMP*

As a conclusion, the more the number of cores is reduced and the more the utility gained by the tolerance-based approach increases compared with that accrued by the SAMP method. Furthermore, the ATMP protocol allows to run more tasks per core when the amount of computing resources decreases.



(a) Comparison on 5 cores



(b) Comparison on 3 cores

Figure 7.9: Absolute utility achieved by each task with SAMP and ATMP

7.3 Chapter Summary

This chapter contains all the experiments made with the scheduling protocols previously introduced in this dissertation.

Section 7.1 contains the discussion, description and evaluation of experiments made with the LBP approaches, which are fixed-priority protocols to

schedule mixed-criticality task sets on uniprocessor platforms. In particular,sub-Section 7.1.3 contains the evaluation of scheduling protocols in a hard real-time settings while sub-Section 7.1.4 contains a study and analysis of performances between LBP and SLBP in the case in which they behave differently, i.e., when tasks have constrained deadlines.

Section 7.2 contains the description and evaluation of experiments made with ATMP, a mixed-criticality protocol based on the TRTCM model [6] for dynamic reallocation of tasks to cores at run-time on multi-core platforms. The results confirm that, compared with the standard approach in which tasks have a fixed and unmodifiable arrival rate, the ATMP allows to keep more tasks allocated in case of sudden faults of cores.

Chapter 8

Conclusions

This chapter concludes my dissertation by providing a summary of its content and a review of all my contributions. It also provides an overview of future enhancements of the protocols introduced both for short and mid-term scheduling decisions.

8.1 Summary of Dissertation

My research work has dealt with the mixed-criticality scheduling with the aim of maximising the number of jobs correctly processed. This dissertation presented a framework to connect the TRTCM mid-term resource optimisation together with the underlying mixed-criticality scheduling protocols. The method is suitable both in case of single and multicore architectures.

Currently, all fixed priority and dynamic priority mixed-criticality scheduling protocols aim to guarantee a correct completion to jobs according to the level of their criticality, established at the design phase. In case of resource shortage the higher criticality jobs execution has to be guaranteed at the expense of lower criticality ones. On the other hand, the TRTCM represents a model in which, in case of resource shortage, the service provided to each task can be smoothly degraded regardless of its criticality and within a certain tolerance interval. The optimisation method finds a trade-off among tasks having different criticality with the final aim to maximise the overall system utility.

The research question that has motivated and guided my research is the following:

Is it possible to improve the state of the art of fixed-priority mixed-criticality scheduling and then combine such protocols with the TRTCM model to optimise the utility of mixed-criticality real-time systems in case of resource shortages?

Such question led me to study and analyse short and mid-term mixed-criticality scheduling protocols both on uniprocessor and multiprocessor architectures. As a result, I have further refined and fractured my main question into the following three sub-questions:

1. **Is it possible to increase the robustness of existing fixed-priority mixed-criticality scheduling protocols?**

2. **Is it possible to connect the TRTCM optimisation problem with the mixed-criticality scheduling protocols?**

3. **Are the utility functions and tolerance ranges of TRTCM a useful means to increase the amount of tasks scheduled in real-time systems with mixed-criticality services?**

To answer to the sub-questions above, first I had to understand how the mixed-criticality scheduling protocols work and the challenges to be faced. With regard to this, Chapter 3 reviewes the research work on mixed-criticality scheduling. It highlights how the challenge of current research is that to design scheduling algorithms to enhance robustness at run-time. A recent method to schedule mixed-criticality task sets is represented by the *Bailout Protocol* (BP) [16, 17]. As previous mixed-criticality protocols, BP schedules set of tasks of low or high criticality, indicated respectively by LO and HI. The novelty of such protocol is that to introduce an effective and fast control mechanism to switch back to the starting execution mode in which all jobs can be scheduled. Although this protocol represents an enhancement if compared with previous scheduling algorithms, it still abandons a large amount of LO jobs at run-time.

The preliminary study of the mixed-criticality scheduling algorithms led to devise the *Lazy Bailout Protocol* (LBP), introduced in Chapter 4. LBP increases the amount of jobs completed within their deadline because LO jobs released in case of resource shortage are inserted in a low-priority queue for background execution during the system idle time rather than being suddenly aborted. LBP works within mixed-criticality environments with all tasks being hard real-time tasks, i.e., each job completion has a value if it occurs within the related deadline. Chapter 4 also presented a variant of LBP, the SLBP protocol, in which HI jobs

have hard deadlines while LO tasks instances have soft deadlines. Such soft real-time jobs could have a maximum tardiness that is bounded and hence does not jeopardise the timeliness of successive instances of the same task. With SLBP, I have also made the existing fixed-priority mixed-criticality methods more suitable in cases in which a tardy completion of LO jobs is better than no result at all.

I further strengthened the robustness of LBP and SLBP by combining them with two complementary scheduling techniques that are usually used to increase the adaptiveness of real-time systems. Such additional strategies are based on the exploitation of the CPU spare capacity, that can be identified either offline or online. The first technique is based on the usage of the offline tuning of the task system while the second is based on the online collection of the gain time. Both methods allows to increase the amount of jobs scheduled without affecting the task set schedulability. The integration of such additional techniques with LBP led to devise LBPG, LBPS and LBPSG while the integration with SLBP led to SLBPG, SLBPS and SLBPSG. Then, Chapter 5 shows, via formal proofs, that each LBP and SLBP-based protocol increases the amount of LO jobs completed within their deadlines if compared with the corresponding BP-based one. This ultimately confirms that LBP and its variants increase the runtime robustness with respect to the existing BP and its derivatives.

After having understood challenges and issues related to the fixed-priority mixed-criticality scheduling on single core architectures, I built a framework to integrate the existing scheduling protocols, including the mixed-criticality ones, with the TRTCM model as required in the second sub-question. With regard to this, Chapter 6 contains the *Adaptive Tolerance-based Mixed-criticality Protocol* (ATMP), a utility and criticality aware strategy to reallocate tasks to cores at runtime in case of sudden unavailability of processing elements. ATMP allows to connect the mid-term TRTCM planning with the underlying mixed-criticality scheduling algorithms. Furthermore, ATMP allows to show how the TRTCM task model minimises the number of lower criticality tasks deallocated from each core while guaranteeing the schedulability of the overall task set. This represents a way to prove the effectiveness of the usage of tolerance intervals and utility functions as means to optimise performances in real-time systems with mixed-criticality services.

Finally, Chapter 7 contains the results and experimental evaluation that give quantitative measurements with regard to the extent of my contributions about fixed-priority scheduling on uniprocessor platforms and utility driven optimisation within the field of mixed-criticality systems. This chapter further confirms the enhancement with respect to the BP-based protocols proved in Chap-

ter 5. Furthermore, experiments show that LBPSG and SLBPSG are the mixed-criticality protocols that achieve better performances, i.e., that mostly increase the amount of LO jobs scheduled without affecting the HI jobs schedulability. Then, this chapter also replies to sub-question 3. I compared ATMP with a mixed-criticality heuristics in which tasks have no tolerance interval to exploit and results confirm that, in case of resource shortage leading to a decrease in the number of active processing elements, the ATMP keeps allocated more tasks to cores by adjusting their arrival rates.

8.2 Research Impact and Application Areas

As explained in Section 8.1, the contributions of my dissertation are represented by the ATMP and LBP/SLBP protocols. ATMP represents a mixed-criticality partitioned scheduling heuristics to allocate and optimise a set of tasks on multicore platforms while the lazy scheduling methods represent algorithms to decide what ready job to execute next on single-core architectures. My research concerns the real-time scheduling in safety-critical systems. In this area, the mixed-criticality scheduling represents a form of scheduling in which faults, either permanent or transient, and more generally resource shortage situations are handled in the scheduling process itself. This is due also to the *Size, Weight and Power* (SWaP) design constraints that have always been important for embedded systems and that sometime make even unsuitable the usage of hardware replication [111]. The contributions proposed in this dissertation permit to increase the robustness of mixed-criticality systems both on single and multi-core architectures since they allow to successfully schedule more jobs in case of unexpected shortage of computational resources. This would reduce the overall system service degradation in all the applications in which timing constraints are part of the functional requirements.

Concrete examples of these applications are represented by distributed and multicore cyber-physical medical systems used in remote telesurgery. Remote telesurgery devices typically consist both of hard real-time image processing and robotic components. An accurate scheduling of tasks is required for both components since such applications are safety-critical, i.e., any communication or processing delay can endanger the patient's life. Therefore precision and accurateness in scheduling such tasks must always be preserved. Typically, images would be collected via camera and sent over a wide area network to a medical device that will schedule all the tasks related to images to provide the results to

the surgeon that remotely can afterwards move a robot to operate the patient.

The tasks involved in the image processing can be the following (each with its own period, deadline and criticality):

- Image filtering: removes noise from images or even enhances brightness or contrast to make the images clearer;
- Object detection: extracts features of interest and highlights the desired object to check;
- Image analysis: extracts and processes some parameters or measurements of interest either from the images or from the objects to be detected;
- Logging: tracks the operations related to all the previous tasks and stores them in the text file.

Figure 8.1 shows an overview of how remote telesurgery works:

Figure 8.1: (Remote surgery) A physician that gets images about the patient via network and then moves a robot for the operation [5]

To build such applications it is necessary to use real-time protocols at Transport layer of the TCP/IP stack and have a very high bandwidth capability but also medical devices equipped with appropriate real-time schedulers that are capable to process tasks by considering both timing requirements and criticalities. In case of problems over the network the medical device can degrade the service provided to LO tasks via LBP/SLBP while in case of sudden unavailability of some core it can reallocate tasks to cores and perform a task periods optimisation via ATMP. Some lower criticality tasks, e.g. logging or even image analysis, might experience a service degradation or be temporarily not provided because deallocated from some core since the mixed-criticality system should schedule first tasks that provide images related to the part of human body to be highlighted. Once the output has been provided, the surgeon can move the robot at the other end of the network to operate the patient.

8.3 Outlook

Within this dissertation, I presented my contributions to enhance the robustness of mixed-criticality scheduling protocols for both short-term and mid-term scheduling decisions. With regard to this, I have introduced scheduling protocols that exploit the system idle time on single core architectures and that permit to optimise the overall system utility by adjusting the overall load on multicore architectures.

Since the timely memory and computing resource allocation to higher criticality tasks has to be guaranteed by assumptions, the current and future challenge will continue to be that to minimise the impact on lower criticality tasks because they could be still important for the mission completion.

Strategies that are possible to devise and to integrate into my research work either to increase robustness at runtime or to extend the scheduling architecture are the following:

- **Usage of a feedback mechanism to reduce the number of LO jobs missing their deadline.**

  As stated in Chapter 2, the feedback mechanism uses information about the actual behaviour of a scheduling system to dynamically adapt the scheduling algorithms such that the intended behaviour is achieved. Unlike ATMP that adjusts the arrival rates of tasks after a change in the number of active processing elements in multi-core systems, the feedback mechanism is used at runtime for short-term scheduling decisions on a single core. The feedback can be integrated within LBP and its variants to dynamically adjust the arrival rate of LO tasks according to the number of their instances that miss their deadlines. Chapter 4 introduced the LBP protocol and Figure 4.1 shows its architecture. The LBP architecture contains two monitors to check respectively the LO and HI jobs execution. LO jobs exceeding their optimistic WCET or released during Bailout and Recovery modes are inserted into a low-priority queue while HI jobs exceeding their optimistic WCET trigger the switch to the Bailout mode. The usage of a feedback mechanism would extend the LO jobs monitor functionality to check also the number of LO jobs missing their deadlines because they are dropped or abandoned. Such information can be used to reduce the overall system load by slowing down the arrival rate of LO tasks with the final aim to guarantee a minimum guaranteed amount of LO jobs completing within their deadlines.

It is worth to notice that previous mixed-criticality scheduling methods, e.g., BP and AMC, are not suitable for an online adaptation of task periods because LO jobs relased during the HI execution modes are suddenly abandoned and this limits the possibility to adjust the LO task periods to properly optimise the system performances. Conversely, the LBP and SLBP approaches permit to freely adjust the task periods according to the specified needs with smaller impact on system performances, i.e., task instances released during the HI modes will be processed during idle instants if there are.

- **Extending the LBP protocol to support multiple criticality levels.**

  Most of mixed-criticality scheduling protocols on uniprocessor platforms devised so far support two levels of criticality, HI and LO with HI being more critical than LO. Introducing the support to schedule set of tasks having more than two criticality levels would permit to schedule jobs at different criticality levels according to different policies in case of resource shortage.

  Unlike the ATMP protocol in which each task exploits a tolerance range to optimise its arrival rate via linear programming [112, 6], the multi-criticality LBP protocol must always ensure the correct completion of highest criticality jobs at their fixed rates while trying to maximise the amount of lower criticality jobs scheduled. Furthermore, lower criticality jobs that have to be guaranteed at different levels of assurance can be processed according to different strategies. As an example, it would be possible to consider a task set containing tasks with three criticality levels, i.e., high (HI), medium (ME) and low (LO). In case any HI job exceeds its optimistic WCET, it would be necessary to ensure its timely completion and meanwhile to find heuristics or best effort strategies to maximise the amount of lower criticality jobs completed within or even after their deadline. It could be possible to adjust the arrival rate of ME jobs via feedback control to minimise the number of deadlines missed or to permit to such jobs to complete even after their deadline. Lastly, no guarantee could be enforced for LO jobs.

- **Integrating the Selective TRTCM approach into the ATMP protocol.**

  The ATMP protocol introduced in Chapter 6 implements the *Retained TRTCM* approach ($TRTCM_{ret}$), in which each task has got a least tolerance

utility under which it cannot be degraded anymore but it will rather be abandoned [6, page 5]. The $TRTCM_{ret}$ is suitable in cases where no service at all is preferrable over poor performances. With $TRTCM_{ret}$, if it is not possible to degrade anymore the least acceptable utility of a lower criticality task but it is still necessary some adjustment to maximise the overall system utility, then such task would be de-allocated from the assigned core.

Conversely, the *Selective TRTCM* ($TRTCM_{sel}$) removes such constraint for lower criticality tasks [6, page 5]. This TRTCM variant is appropriate for systems that contain tasks that are optional for mission completion. Through this extra flexibility it is possible to further degrade the service provided to lower criticality tasks and to allocate more resources to higher criticality tasks. In case of just two criticality levels indicated by LO and HI, this would lead to slowing down LO task arrival rates even below their least tolerance bounds.

Bibliography

[1] R. Kirner, "A uniform model for tolerance-based real-time computing," in *17th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2014, Reno, NV, USA, June 10-12, 2014*, pp. 9–16, 2014.

[2] H. Kopetz, *Real-Time Systems - Design Principles for Distributed Embedded Systems*. Real-Time Systems Series, Springer, second ed., 2011.

[3] M. Rouhifar, "A survey on scheduling approaches for hard real-time systems," *International Journal of Computer Applications (IJCA)*, vol. 131, no. 17, 2015.

[4] S. Afshar, "Scheduling and resource sharing in multiprocessor real-time systems." `https://www.slideshare.net/knowdiff/presentations`, July 2015. Slides on www.slideshare.net.

[5] D. Obenshain and T. Tantillo, "Remote telesurgery." `http://www.cnds.jhu.edu/~dano/RemoteTelesurgery.pdf`, 2018.

[6] R. Kirner, S. Iacovelli, and M. Zolda, "Optimised adaptation of mixed-criticality systems with periodic tasks on uniform multiprocessors in case of faults," in *Proc. 11th IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'15)*, (Auckland, New Zealand), April 2015.

[7] RTCA, "Software considerations in airborne systems and equipment certification." RTCA/DO-178B, December 2011.

[8] ISO/DIS, "Road vehicles – functional safety." ISO/DIS standard 26262, November 2011. International Standard.

[9] I. E. Commission, "Functional safety of electrical / electronic / programmable electronic safety-related systems." IEC standard 61508, 1998.

[10] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer Publishing Company, Incorporated, 3rd ed., 2011.

[11] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *28th IEEE Real-Time System Symposium (RTSS 2007)*, pp. 239–243, December 2007.

[12] S. Baruah, "Mixed-criticality scheduling theory: Scope, promise, and limitations," *IEEE Design & Test*, vol. 35, no. 2, pp. 31–37, 2018.

[13] D. de Niz, K. Lakshmanan, and R. R. Rajkumar, "On the scheduling of mixed-criticality real-time task sets," in *RTSS '09 Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, pp. 291–300, December 2009.

[14] A. Thekkilakattil, A. Burns, R. Dobrin, and S. Punnekkat, "Mixed criticality systems: Beyond transient faults," in *Proceedings of the third International Workshop on Mixed Criticality Systems (WMC)*, (San Antonio, Texas, USA), December 2015.

[15] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckman, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrom, "The worst-case execution time problem - overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, April 2008.

[16] I. Bate, A. Burns, and R. I. Davis, "A bailout protocol for mixed criticality systems," in *27th Euromicro Conference on Real-Time Systems*, 2015.

[17] I. Bate, A. Burns, and R. I. Davis, "An enhanced bailout protocol for mixed criticality embedded software," *IEEE Transactions on Software Engineering*, vol. 43, pp. 298–320, Apr. 2017.

[18] S. Tobuschat, P. Axer, R. Ernst, and J. Diemer, "IDAMC: A noc for mixed criticality systems," in *IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2013, Taipei, Taiwan, August 19-21, 2013*, pp. 149–156, August 2013.

[19] B. Motruk, J. Diemer, R. Buchty, R. Ernst, and M. Berekovic, "IDAMC: A many-core platform with run-time monitoring for mixed-criticality," in *IEEE 14th International Symposium on High-Assurance Systems Engineering (HASE)*, pp. 24–31, IEEE Computer Society, October 2012.

[20] H. Su, D. Zhu, and D. Mossé, "Scheduling algorithms for elastic mixed-criticality tasks in multicore systems," in *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*, IEEE, August 2013.

[21] 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)*, (Taipei, Taiwan), pp. 1–6, August 2013.

[22] R. Kirner, "Ingredients for the specification of mixed-criticality real-time systems," in *17th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2014, Reno, NV, USA, June 10-12, 2014*, pp. 269–275, 2014.

[23] G. Lipari and L. Palopoli, "Real-time scheduling: from hard to soft real-time systems," *CoRR*, vol. abs/1512.01978, 2015.

[24] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS '98, (Washington, DC, USA), pp. 4–, IEEE Computer Society, 1998.

[25] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of Association for Computing Machinery*, vol. 20, pp. 46–61, January 1973.

[26] S. Baruah, M. Bertogna, and G. Buttazzo, *Multiprocessor Scheduling for Real-Time Systems*. Embedded Systems, Springer, 2015.

[27] R. I. Davis, L. C.-G. Liliana, M. Bertogna, and A. Burns, "A review of priority assignment in real-time systems," *Journal of System Architecture*, vol. 65, pp. 64–82, April 2016.

[28] S. Baruah and A. Burns, "Sustainable scheduling analysis," in *27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pp. 159–168, December 2006.

[29] G. C. Buttazzo, "Rate monotonic vs. edf: Judgment day," *Real-Time Systems*, vol. 29, pp. 5–26, January 2005.

[30] J. Y. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance Evaluation*, vol. 2, pp. 237–250, December 1982.

[31] J. Lehoczky, "Fixed priority scheduling of periodic task sets with arbitrary deadlines," in *Proceedings 11th Real-Time Systems Symposium (RTSS'90)*, December 1990.

[32] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings, "Hard real-time scheduling: The deadline-monotonic approach," in *in Proceedings IEEE Workshop on Real-Time Operating Systems and Software (RTOSS)*, pp. 133–137, May 1991.

[33] N. C. Audsley, A. Burns, M. M. Richardson, K. Tindell, and A. J. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, 1993.

[34] J. P. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: Exact characterization and average case behavior," in *Proceedings of the Real-Time Systems Symposium - 1989, Santa Monica, California, USA, December 1989*, pp. 166–171, 1989.

[35] M. L. Dertouzos, "Control robotics: The procedural control of physical processes.," in *IFIP Congress*, pp. 807–813, 1974.

[36] E. D. Jensen, C. D. Locke, and H. Tokuda, "A time-driven scheduling model for real-time operating systems," in *6th IEEE Real-Time Sys. Symp. (RTSS'85)*, pp. 112–122, December 1985.

[37] P. Li, B. Ravinan, and E. D. Jensen, "Adaptive time-critical resource management using time/utility functions: Past, present and future," in *28th Annual International Computer Software and Applications Conference*, 2004.

[38] B. Ravindran, E. D. Jensen, and P. Li, "On recent advances in time/utility function real-time scheduling and resource management," in *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, May 2005.

[39] A. Colin and S. M. Petters, "Experimental evaluation of code properties for WCET analysis," in *Proceedings of the 24th IEEE Real-Time Systems*

Symposium (RTSS 2003), 3-5 December 2003, Cancun, Mexico, pp. 190–199, 2003.

[40] N. C. Audsley, R. I. Davis, and A. Burns, "Mechanisms for enhancing the flexibility and utility of hard real-time systems," in *RTSS*, pp. 12–21, IEEE Computer Society, 1994.

[41] R. I. Davis, *On Exploiting Spare Capacity in Hard Real-Time Systems*. PhD thesis, Department of Computer Science, University of York, July 1995.

[42] D. Haban and K. G. Shin, "Application of real-time monitoring to scheduling tasks with random execution times," *IEEE Trans. Software Eng.*, vol. 16, no. 12, pp. 1374–1389, 1990.

[43] E. Y. Hu, A. J. Wellings, and G. Bernat, "Gain time reclaiming in high performance real-time java systems," in *ISORC*, pp. 249–256, IEEE Computer Society, 2003.

[44] S. Punnekkat, R. Davis, and A. Burns, "Sensitivity analysis of real-time task sets," in *Advances in Computing Science - ASIAN'97: Third Asian Computing Science Conference Kathmandu, Nepal, December 9-11, 1997 Proceedings*, (Berlin, Heidelberg), pp. 72–82, Springer Berlin Heidelberg, 1997.

[45] S. Vestal, "Fixed-priority sensitivity analysis for linear compute time models," *IEEE Transactions on Software Engineering*, vol. 20, no. 4, pp. 308–317, 1994.

[46] D. I. Katcher, H. Arakawa, and J. K. Strosnider, "Engineering and analysis of fixed priority schedulers," *IEEE Transactions on Software Engineering*, vol. 19, pp. 920–934, Sept. 1993.

[47] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, pp. 35:1–35:44, October 2011.

[48] T. P. Baker, "A comparison of global and partitioned edf schedulability tests for multiprocessors," tech. rep., Florida State University, Dept. of Computer Science, 2005.

[49] S. K. Baruah and J. Goossens, "Rate-monotonic scheduling on uniform multiprocessors," in *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.*, May 2003.

[50] T. P. Baker and S. K. Baruah, "Sustainable multiprocessor scheduling of sporadic task systems," in *Proceedings of the 2009 21st Euromicro Conference on Real-Time Systems*, ECRTS '09, (Washington, DC, USA), pp. 141–150, IEEE Computer Society, July 2009.

[51] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, first edition ed., 1979.

[52] B. Andersson and J. Jonsson, "Fixed-priority preemptive multiprocessor scheduling: To partition or not to partition," in *Proceedings of the Seventh International Conference on Real-Time Systems and Applications*, RTCSA '00, (Washington, DC, USA), pp. 337–, IEEE Computer Society, 2000.

[53] N. Guan, P. Ekberg, M. Stigge, and W. Yi, "Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems," in *Proceedings of the 2011 IEEE 32nd Real-Time Systems Symposium*, RTSS '11, (Washington, DC, USA), pp. 13–23, IEEE Computer Society, 2011.

[54] S. K. Baruah, "Certification-cognizant scheduling of tasks with pessimistic frequency specification," in *7th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pp. 31–38, IEEE, June 2012.

[55] A. Burns and R. I. Davis, "A survey of research into mixed criticality systems," *ACM Comput. Surv.*, vol. 50, pp. 82:1–82:37, November 2017.

[56] A. Burns and R. I. Davis, "A survey of research into mixed criticality systems," *ACM Comput. Surv.*, vol. 50, pp. 82:1–82:37, Nov. 2017.

[57] A. Wasicek, C. El–Salloum, and H. Kopetz, "A system-on-a-chip platform for mixed-criticality applications," in *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pp. 174–188, IEEE, May 2010.

[58] A. Burns and S. Baruah, *Dependable and Historic Computing*, vol. 6875 of *Lecture Notes in Computer Science*, ch. Timing Faults and Mixed Criticality Systems, pp. 147–166. Springer, 2011.

[59] S. Baruah and B. Chattopadhyay, "Response-time analysis of mixed criticality systems with pessimistic frequency specification," in *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, August 2013.

[60] S. Baruah, "Schedulability analysis of mixed-criticality systems with multiple frequency specifications," in *Proceedings of the 13th International Conference on Embedded Software, EMSOFT 2016*, Association for Computing Machinery, Inc", 10 2016.

[61] A. Burns and R. I. Davis, "Mixed criticality on controller area network," in *25th Euromicro Conference on Real-Time Systems, ECRTS 2013, Paris, France, July 9-12, 2013*, pp. 125–134, 2013.

[62] S. Baruah and A. Burns, "Implementing mixed criticality systems in ada," in *Proceedings of the 16th Ada-Europe International Conference on Reliable Software Technologies*, Ada-Europe'11, (Berlin, Heidelberg), pp. 174–188, Springer-Verlag, 2011.

[63] A. Crespo, A. Alonso, M. Marcos, J. A. de la Puente, and P. Balbastre, "Mixed criticality in control systems," in *Proc. 19th World Congress, The International Federation of Automatic Control*, (Cape Town, South Africa), August 2014.

[64] O. Cros, L. George, and X. Li, "A protocol for mixed-criticality management in switched ethernet networks," in *Proceedings of the third International Workshop on Mixed Criticality Systems (WMC)*, (San Antonio, Texas, USA), December 2015.

[65] R. Ernst and M. D. Natale, "Mixed criticality systems - a history of misconceptions," in *IEEE Design and Test*, vol. 33, pp. 65–74, IEEE Design and Test, October 2016.

[66] A. Esper, G. Nelissen, V. Nélis, and E. Tovar, "How realistic is the mixed-criticality real-time system model?," in *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, RTNS '15, (New York, NY, USA), pp. 139–148, ACM, 2015.

[67] R. Ernst and M. D. Natale, "Mixed criticality systems - a history of misconceptions?," *IEEE Design & Test*, vol. 33, no. 5, pp. 65–74, 2016.

[68] P. Graydon and I. Bate, "Safety assurance driven problem formulation for mixed-criticality scheduling," in *Proceedings of the Workshop on Mixed-Criticality Systems*, pp. 19–24, 2013.

[69] S. K. Baruah, A. Burns, and R. I. Davis, "Response-time analysis for mixed criticality systems," in *Proceedings of the 2011 IEEE 32Nd Real-Time Systems Symposium*, RTSS '11, (Washington, DC, USA), pp. 34–43, IEEE Computer Society, November 2011.

[70] A. Burns and R. I. Davis, "Response time analysis for mixed criticality systems with arbitrary deadlines," in *5th International Workshop on Mixed Criticality Systems (WMC 2017)*, December 2017.

[71] L. Li, R. Li, L. Huang, R. Wu, and L. Zeng, "A new rta based scheduling algorithm for mixed-criticality systems," in *Proc. of 16th International Conference on Computational Science and Engineering*, IEEE, December 2013.

[72] Z. Guo, S. Sruti, B. C. Ward, and S. Baruah, "Sustainability in mixed-criticality scheduling," in *IEEE Real-Time Systems Symposium, RTSS 2017, Paris, France, December 5-8, 2017*, pp. 24–33, December 2017.

[73] F. Zhang, K. Szwaykowska, W. Wolf, and V. Mooney, "Task scheduling for control oriented requirements for cyber-physical systems," in *Proceedings of the 2008 Real-Time Systems Symposium*, RTSS '08, (Barcelona, Spain), pp. 47–56, IEEE Computer Society, December 2008.

[74] H. Su and D. Zhu, "An elastic mixed-criticality task model and its scheduling algorithm," in *Conference on Design, Automation and Test in Europe (DATE)*, pp. 147–152, IEEE, March 2013.

[75] A. Burns and S. Baruah, "Towards a more practical model for mixed criticality systems," in *Proc. of 1st International Workshop on Mixed Criticality Systems (WMC)*, pp. 1–6, December 2013.

[76] S. Baruah, A. Burns, and R. Davis, "An extended fixed priority scheme for mixed criticality systems," in *ReTiMiCS, RTCSA* (L. George and G. Lipari, eds.), pp. 18–24, 2013.

[77] A. Zuhily and A. Burns, "Optimal (d-j)-monotonic priority assignment," *Information Processing Letters*, vol. 103, pp. 247–250, 2007.

[78] T. Fleming and A. Burns, "Incorporating the notion of importance into mixed criticality systems," in *Proc. of 2nd International Workshop on Mixed Criticality Systems (WMC)*, pp. 33–38, December 2014.

[79] D. Prasad, A. Burns, and M. Atkins, "The valid use of utility in adaptive real-time systems," *Real-Time Systems*, vol. 25, pp. 277–296, September 2003.

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

[81] O. Gettings, S. Quinton, and R. I. Davis, "Mixed criticality systems with weakly-hard constraints," in *RTNS '15: Proceedings of the 23rd International Conference on Real Time and Networks Systems*, (New York, NY, USA), pp. 237–246, ACM, November 2015.

[82] G. Bernat, A. Burns, and A. Llamosi, "Weakly hard real-time systems," *IEEE Trans. Comput.*, vol. 50, pp. 308–321, Apr. 2001.

[83] P. Ramanathan and M. Hamdaoui, "A dynamic priority assignment technique for streams with (m, k)-firm deadlines," *IEEE Trans. Comput.*, vol. 44, pp. 1443–1451, December 1995.

[84] G. Koren and D. Shasha, "Skip-over: Algorithms and complexity for overloaded systems that allow skips," in *Proceedings of IEEE Real-Time System Symposium (RTSS)*, pp. 110–117, IEEE, December 1995.

[85] G. Frehse, A. Hamann, S. Quinton, and M. Woehrle, "Formal analysis of timing effects on closed-loop properties of control software," in *Proceedings of the 2011 IEEE 32Nd Real-Time Systems Symposium*, RTSS '14, pp. 53–62, IEEE, December 2014.

[86] F. Santy, L. George, P. Thierry, and J. Goossens, "Relaxing mixed-criticality scheduling strictness for task sets scheduled with fp," in *Proc. in 24th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 155–165, IEEE, July 2012.

[87] 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 - 19th Annual European Symposium, Saarbrücken, Germany, September 5-9, 2011. Proceedings*, pp. 555–566, 2011.

[88] S. K. 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 *24th Euromicro Conference on Real-Time Systems, ECRTS 2012, Pisa, Italy, July 11-13, 2012*, pp. 145–154, 2012.

[89] H. Su and D. Zhu, "An elastic mixed-criticality task model and its scheduling algorithm," in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, (San Jose, CA, USA), pp. 147–152, EDA Consortium, 2013.

[90] G. C. Buttazzo, G. Lipari, and L. Abeni, "Elastic task model for adaptive rate control," in *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS '98, (Washington, DC, USA), IEEE Computer Society, 1998.

[91] R. Obermaisser and D. Weber, "Architectures for mixed-criticality systems based on networked multi-core chips," in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation, ETFA 2014, Barcelona, Spain, September 16-19, 2014*, pp. 1–10, 2014.

[92] V. Legout, M. Jan, and L. Pautet, "An off-line multiprocessor real-time scheduling algorithm to reduce static energy consumption," in *First Workshop on Highly-Reliable Power-Efficient Embedded Designs (HARSH)*, (Shenzhen, China), pp. 7–12, February 2013.

[93] A. Thekkilakattil, R. Dobrin, and S. Punnekkat, "Mixed criticality scheduling in fault-tolerant distributed real-time systems," in *2014 International Conference on Embedded Systems (ICES)*, pp. 92–97, July 2014.

[94] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory access control in multiprocessor for real-time systems with mixed criticality," in *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, ECRTS '12, (Washington, DC, USA), pp. 299–308, IEEE Computer Society, 2012.

[95] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele, "Scheduling of mixed-criticality applications on resource-sharing multicore systems," in

Proceedings of the Eleventh ACM International Conference on Embedded Software, EMSOFT '13, (Piscataway, NJ, USA), pp. 17:1–17:15, IEEE Press, 2013.

[96] R. Trüb, G. Giannopoulou, A. Tretter, and L. Thiele, "Implementation of partitioned mixed-criticality scheduling on a multi-core platform," *ACM Trans. Embed. Comput. Syst.*, vol. 16, pp. 122:1–122:21, September 2017.

[97] A. Burns, T. Fleming, and S. Baruah, "Cyclic executives, multi-core platforms and mixed criticality applications," in *Proceedings of the 2015 27th Euromicro Conference on Real-Time Systems*, ECRTS '15, (Washington, DC, USA), pp. 3–12, IEEE Computer Society, 2015.

[98] D. S. Johnson, *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, Department of Mathematics, June 1973.

[99] V. Izosimov and E. Levholt, "Mixed criticality metric for safety-critical cyber-physical systems on multi-core architectures," in *Proceedings of the 4th Workshop On Manufacturable and Dependable Multicore Architectures at Nanoscale (MEDIAN'15)*, MEDIAN '15, March 2015.

[100] RTCA, "Software considerations in airborne systems and equipment certification." RTCA/DO-178B, 1992.

[101] H. Su, D. Zhu, and S. Brandt, "An elastic mixed-criticality task model and early-release edf scheduling algorithms," *ACM Transactions on Design Automation Electronic Systems (TODAES)*, vol. 22, pp. 28:1–28:25, December 2016.

[102] R. Schneider, D. Goswami, A. Masrur, and S. Chakraborty, "Qoc-oriented efficient schedule synthesis for mixed-criticality cyber-physical systems," in *Proceeding of the 2012 Forum on Specification and Design Languages (FDL)*, September 2012.

[103] R. Schneider, D. Goswami, A. Masrur, M. Becker, and S. Chakraborty, "Multi-layered scheduling of mixed-criticality cyber-physical systems," *Journal of Systems Architecture*, vol. 59, pp. 1215–1230, November 2013.

[104] 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, CIT '10, (Washington, DC, USA), pp. 1864–1871, IEEE Computer Society, 2010.

[105] D. Tamas-Selicean and P. Pop, "Optimization of time-partitions for mixed-criticality real-time distributed embedded systems," in *14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops, ISORC Workshops 2011, Newport Beach, CA, USA, March 28-31, 2011*, pp. 1–10, 2011.

[106] E. Bini, M. D. Natale, and G. C. Buttazzo, "Sensitivity analysis for fixed-priority real-time systems," *Real-Time Systems*, vol. 39, no. 1-3, pp. 5–30, 2008.

[107] "AUTOSAR development standards partnerships." `www.autosar.org`, 2018. Automotive Open System Architecture.

[108] I. E. Commission, "Functional safety of electrical, electronic, programmable electronic safety-related systems," 2010.

[109] E. Parliament and the Council of the European Union, "Directive (eu) 2016/1148 concerning measures for a high common level of security of network and information systems across the union," 2016.

[110] N. C. S. Centre, "Nis-directive: Top level objectives." `https://www.ncsc.gov.uk/guidance/nis-directive-top-level-objectives`, 2018.

[111] C. Lee, H. Kim, H. woo Park, S. Kim, H. Oh, and S. Ha, "A task remapping technique for reliable multi-core embedded systems.," in *IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)* (T. Givargis and A. Donlin, eds.), pp. 307–316, ACM, 2010.

[112] S. Iacovelli, R. Kirner, and C. Menon, "ATMP: an adaptive tolerance-based mixed-criticality protocol for multi-core systems," in *13th IEEE International Symposium on Industrial Embedded Systems, SIES 2018, Graz, Austria, June 6-8, 2018*, pp. 1–9, 2018.