Allocation and Optimisation of Mixed Criticality Cyclic Executives

Thomas David Fleming

Doctor Of Philosophy

University Of York

Computer Science

June 2017

Abstract

Incorporating applications of differing levels of criticality onto the same platform in an efficient manner is a challenging problem. Highly critical applications require stringent verification and certification while lower criticality work may seek to make greater use of modern processing power with little to no requirement for verification. Much study into mixed criticality systems has considered this issue by taking scheduling paradigms designed to provide good platform utilisation at the expense of predictability and attempting to provide mechanisms that will allow for the verification of higher criticality work. In this thesis we take the alternative approach, we utilise a cyclic executive scheduler. Such schedulers are used extensively in industrial practice and provide very high levels of determinism making them a strong choice for applications with strict certification requirements. This work provides a platform which supports the highly critical work, alongside work of lower criticalities in a cyclic executive context. The aim being to provide a near-future platform which is able to support existing legacy highly critical software alongside newer less critical software which seeks to utilise multi-core architectures. One of the fundamental challenges of designing a system for a static scheduler is the allocation of applications/tasks to the cores and, in the case of cyclic executives, minor cycles of the system. Throughout this work we explore task allocation, we make extensive use of Linear Programming to model and allocate work. We suggest a limited task splitting technique to aid in system design and allocation. Finally, we propose two ways in which an allocation of work might be optimised to meet some design goal. This thesis proposes a scheduling policy for mixed criticality multi-core systems using a cyclic executive scheduler and explores the design, allocation and optimisation of such a system.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to acknowledge the support and guidance of my supervisor Alan Burns whose knowledge, expertise and experience have been invaluable throughout my PhD. I would like to thank my assessor Leandro Indrusiak for the helpful comments, guidance and always encouraging me to improve my work. I would like to thank and acknowledge the support of BAE Systems as my CASE award sponsors, particularly Paul Moxon as my industrial supervisor and Jane Fenn for helping hold everything together.

Finally, I would like to thank Agnieszka, my parents David and Lesley, and Archie for supporting and keeping me going throughout.

Declaration

I declare that this thesis is a presentation of original work undertaken at the University of York from 2013 - 2017. This work has not been previously presented for an award at this, or any other, University. All work in this thesis is the sole work of the author, unless otherwise stated, all sources are acknowledged. Some of the work in this thesis has been previously published in the following papers.

- T. Fleming and A. Burns. Incorporating the notion of importance into mixed criticality systems. In *Proceedings of Workshop on Mixed Criticality, IEEE Real-Time Systems Symposium (RTSS)*, pages 33-38, 2014

- A. Burns, T. Fleming, and S. Baruah. Cyclic executives, multi-core platforms and mixed criticality applications. In *2015 Euromicro Conference on Real-Time Systems*, pages 3-12, July 2015

- T. Fleming and A. Burns. Investigating mixed criticality cyclic executive schedule generation. In *Proceedings of Workshop on Mixed Criticality, IEEE Real-Time Systems Symposium (RTSS)*, 2015

- T. Fleming, S. Baruah, and A. Burns. Improving the schedulability of mixed criticality cyclic executives via limited task splitting. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, RTNS'16, pages 277-286, New York, NY, USA, 2016. ACM

- T. Fleming, H-M. Huang, A. Burns, C. Gill, S. Baruah, and C. Lu. Corrections to and discussion of "implementation and evaluation of mixed-criticality scheduling approaches for sporadic tasks". *ACM Trans. Embed. Comput. Syst.*, 16(77), April 2017

- T. Fleming and A. Burns. Utilising asymmetric parallelism in multi-core mcs implemented via cyclic executives. In *Proceedings of Workshop on Mixed Criticality, IEEE Real-Time Systems Symposium (RTSS)*, 2016

- C. Deutschbein, T. Fleming, A. Burns, and S. Baruah. Multi-core cyclic executives for saftey-critical systems. In K. G. Larsen, O. Sokolsky, and J. Wang, editors, Proc. *Dependable Software Engineering. Theories, Tools and Applications*, SETTA, pages 94-109. Spring International Publishing, 2017

Chapter 1

Introduction

Real-time applications are becoming increasingly complex due to advancements in both technology and hardware. The increase in complexity and the available resources is leading to a demand for more centralised architectures. Previously, different functionality would often be handled on many different nodes. If some functionality is deemed more important than another it is physically separated in hardware. The advent of powerful consolidated architectures creates a situation where it is desirable to execute applications of multiple levels of importance or Criticality on the same platform, such a system is known as a Mixed Criticality (MC) system.

1.1 Mixed Criticality

A mixed criticality system can be defined as: A system which contains applications of two or more different levels of criticality [81]. Examples of mixed criticality systems include:

- An Un-manned Aerial Vehicle (UAV). Alongside the highly critical flight controls, such a system must also include navigation and mission systems. This might even extend to computer vision applications when a camera is employed for image recognition and/or weapons management. In this scenario the advantage of mixed criticality is in increasing capability while reducing space, weight and power requirements.

- Modern Cars. Modern cars look to reduce the number of processing nodes and may potentially wish to execute applications which manage the anti-lock brakes (ABS) alongside less critical applications such as climate control. The advantage for the automotive domain is reduced hardware cost in mass production.

Many real-time applications require certification in order to be deployed, for example, systems in the aerospace or automotive domain. A fundamental argument must be made about the function of an application and its sufficient isolation from other components in the system. Previously, this was largely achieved by physically separating functions with different levels of criticality. Criticality levels are often defined by the industrial domain, aerospace might use DAL (Design Assurance Levels[1]) and the automotive industry might use ASIL (Automotive Safety Integrity Levels[2]). However, when presented with a situation where all system components share the same hardware, new mechanisms are required to guarantee suitable isolation and safety.

The challenges of isolation are pushed further when the architecture of modern hardware is considered. Increasing the speed of a platform is no longer about an increase in clock speed, multi-core CPUs are the method of choice to improve performance. Multiple CPUs introduces the potential for parallel execution of work which is problematic for real-time applications. Real-time applications often require strict verification, the addition of multiple cores with many more points of interference is challenging. In addition, from the mixed criticality perspective it is important to ensure that no work of a higher criticality level may suffer interference from that of a lower level (or at least any interference suffered is strictly bounded).

1.2 Cyclic Executive

Cyclic Executives (CE) are a well established and understood scheduling technique used extensively in industry. Their popularity is due to their highly deterministic nature, thus providing a scheduling policy whose execution is easy to verify and

[1] https://www.rtca.org/, http://www.eurocae.net/
[2] https://www.iso.org/

therefore certify. The basic structure of a cyclic executive is as follows:

All execution is contained within a major cycle of length T^M. Once completed, the major cycle repeats execution from the start in a cyclic manner. Within the major cycle are a number of minor cycles of length T^F, typically these are the same length. This structure imparts some constraints on the cyclic executive:

- Tasks must have periods that are multiples of the minor cycle.
- Tasks must have periods that are no greater than the major cycle.
- Tasks must have deadlines greater than or equal to the minor cycle.

The cyclic executive structure is illustrated in Figure 1.1:

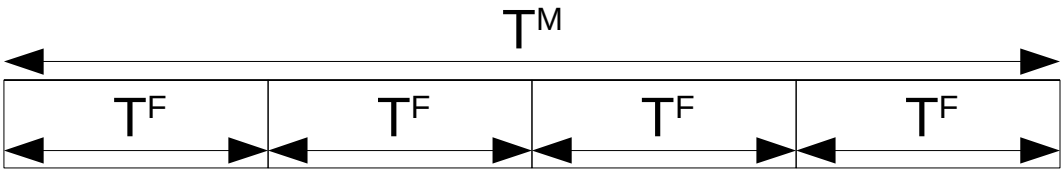


Figure 1.1: An illustration of the Cyclic Executive Structure.

Figure 1.1 illustrates the cyclic executive structure with a major cycle comprised of four minor cycles. While the requirements for a cyclic executive system are clearly restrictive, requiring tasks to adhere to a number of constraints, their advantage lies in their high level of determinism. As such, cyclic executives are favoured in hard real-time scenarios where predictability and the verification of execution are paramount.

Mixed criticality functionality may be introduced to cyclic executives via the temporal separation of execution of differing criticality levels. This is done such that no work of one level of criticality may interfere with another. Criticality levels may then be executed, highest first, until all work has completed. While this complete separation seems restrictive when compared to more dynamic approaches, it allows mixed criticality functionality in cyclic executive systems while maintaining their reputation for predictability and determinism.

While cyclic executives are known for a high level of determinism, they are also notorious for poor overall system utilisation. Key to understanding and tackling this issue is an effective means of task allocation. Is is clear from the structure of the cyclic executive that this is an important and difficult problem.

1.3 Linear Programming

A mixed criticality multi-core cyclic executive presents a complex allocation problem. Such a problem is akin to the well documented bin-packing problem. A standard approach might be to consider a heuristic allocation solution such as worst fit to allocate work to cores and minor cycles. However, given the NP-Hard nature of MC scheduling [59] a more intensive but optimal approach can be considered. Linear Programming (LP) [32] is a means of optimising, either maximising or minimising some linear function. The origins of the current formulation can be traced back to WWII where it was primarily used to solve logistical problems. Linear Programming may be used to determine an optimal assignment of work to cores and minor cycles. In this case optimality is defined such that if an allocation exists, the LP will find it.

An Integer Linear Program (ILP) [32] may be defined and run as a feasibility test (with no optimisation goal required) in order to determine an optimal allocation of tasks to minor cycles and CPU cores. Being a feasibility test it becomes an efficient allocation process. Utilisation of a platform may be increased further from the initial ILP model by making use of Mixed Linear Programming (MLP). Mixed Linear Programs use both integer and continuous variables to represent tasks which may and may not split. Splitting even a small number of tasks can greatly increase the overall utilisation and schedulability of a system. Finally, we may use linear programming tools to optimise an allocation toward a certain goal, giving a system designer freedom to investigate multiple allocations.

1.4 Thesis Hypothesis and Contributions

Mixed Criticality Cyclic Executives provide an attractive platform for highly critical near-future systems. The challenge of allocating tasks to the platform, providing support for design and aiding in allocation optimisation can be achieved through the use of Linear Programming.

The allocation and optimisation of mixed criticality cyclic executives is fundamental to the design of such systems. Linear Programming tools can be used to address this problem. Powerful solvers exist (such as Gurobi [45]) which are able

to effectively and efficiently solve the allocation problems. The key contributions of this work are as follows:

- Our work provides a means of supporting mixed criticality workloads on multi-core platforms scheduled as a cyclic executive. We take a conservative approach (in-line with industrial practice), completely separating criticality levels and utilise a barrier in order to achieve improved schedulability. We assume a simplistic multi-core platform (platform features such as cache, memory, pipeling and multithreading are not considered), all tasks are assumed to be independent. Tasks will always execute as allocated (interrupts are not considered) and any context switching costs are assumed to be negligible.

- In Chapter 3 we consider the allocation of tasks to a multi-core cyclic executive utilising a number of heuristic approaches and a linear programming solver. We show that while heuristics are sufficient for a simpler case, the linear programming solver outperforms them as the complexity of the allocation is increased.

- Evaluations undertaken in Chapters 3, 4 and 5 illustrate that our Linear Programming based allocation approach is effective and reasonably efficient as an offline tool for system prototyping and design.

- We develop a Linear Programming model to represent our cyclic executive allocation problem in Chapter 3.

- We extended the Linear Programming model in Chapter 4 to provide limited task splitting. We show that the splitting of both HI and LO criticality tasks can improve the overall scheduability of a system. Simply splitting 1 or 2 larger tasks is shown to have a big impact of the feasibility of a task set.

- In Chapter 5 we propose two possible optimisations and apply them to our linear programming model. Firstly we show that our model can be used to maximise the space capacity available during the execution of a particular criticality level. Secondly we present an optimisation which seeks to reduce the number of cores utilised by HI criticality tasks with a view to reducing overheads and the cost of verification and certification.

- In Chapter 6 we apply the approaches from Chapters 3, 4 and 5 to an industrial case study based on a real avionics system provided by BAE Systems. We describe and discuss the nature of the case study and illustrate that our techniques can be successfully and efficiently applied to a real-world example.

1.5 Thesis Structure

The remainder of this thesis is structured as follows:

- Chapter Two provides an overview of work related to the subject of the thesis. A broad understanding of mixed criticality literature is presented alongside more detailed focus on work more relevant to the thesis.
- Chapter Three begins the technical work by presenting an investigation into standard task allocation and the development of a Linear Programming model to facilitate this.
- Chapter Four considers the notion of limited task splitting to aid in system design. Mixed Linear Programming is utilised, this sees the solver making decisions about where and how tasks may be split.
- Chapter Five investigates how the optimisation power of Linear Programming may be used to influence the allocation of tasks to achieve a specific goal.
- Chapter Six applies the techniques developed in chapters three, four and five to an industrial case study.
- Chapter Seven concludes the thesis.

Chapter 2

Related Work

A mixed criticality system can be defined as *A system with applications of two or more levels of criticality executing on the same hardware platform* [81]. The criticality level of an application depends on the level of rigour required for its verification and certification. Standards defining levels of rigour are already defined in some industrial domains. Safety Integrity Levels (SIL) (defined in IEC 61580[1]) are used across a wide number of domains including; nuclear, railway and automotive. ASIL (Automotive Safety Integrity Levels) (defined in ISO 26262[2]) are a similar notion specifically tailored to the automotive domain. Aerospace also makes use of Design Assurance Levels (DAL), similar to other levels specified above, these are defined in the DO-178B/C[3] safety standard. Each of these level definitions provide 4 or 5 different levels defined by failure rates, with the highest level often requiring a rate of at most 10^{-9} failures per hour and the lowest level often being best effort. Criticality levels are a more general term for such standards, they are a notion of the required isolation and assurance for a particular application. A mixed criticality system seeks to execute applications of two or more levels on a single platform, the challenge is to maintain the required level of assurance for each.

Much of the current mixed criticality literature can trace its routes back to the seminal paper by Vestal [81] in 2007. Vestal presented a mixed criticality task model which became the backbone for the majority of subsequent work. Vestal's task model is based on the assumption that each given level of criticality comes

[1] http://www.61508.org/
[2] https://www.iso.org/
[3] https://www.rtca.org/, http://www.eurocae.net/

with its own requirements for verification and analysis. As such, a task may have a Worst Case Execution Time (WCET) verified at its level of criticality and each level below. Crucially, Vestal assumes that given two levels of criticality HI and LO (where HI is a higher criticality level than LO), the WCETs derived at each level for task τ_i follow the rule $C_i(HI) \geq C_i(LO)$. In other words, the WCET of a task at a higher criticality level is always greater than or equal to the WCET at a lower criticality level. This relationship is due to increasingly strigent certification and verification processes applied as the level of criticality is incresed, resulting it a higher level of pesimmism and larger WCETs. The remainder of Vestal's work focuses on a Fixed Priority (FP) model and provides the first mixed criticality analysis. This analysis however, assumes that all applications in a system are verified to a single level of criticality, therefore the highest level of criticality. In addition, he notes that Rate Monotonic [61] and Deadline Monotonic [5] priority orders are not optimal for the mixed criticality case. Key to the work in this thesis is the definition of the initial mixed criticality model with the notion of a WCET for each criticality level.

This chapter provides a review of related mixed criticality work. We begin a definition of the mixed criticality model in Section 2.1. We provide a general overview of mixed criticality scheduling on uni-processors in 2.2. Moving on to multi-processor scheduling in Section 2.3. Section 2.4 discusses the use of linear programming in a mixed criticality context. Section 2.5 discusses the use of cyclic executives, both in their simple and mixed criticality forms. Section 2.6 discusses the use of a barrier mechanism. Finally, Section 2.7 summarises the chapter.

2.1 The System Model

This section defines the general mixed criticality model employed by much of the literature. A mixed criticality system can be defined as a finite set of components K. Each K is assigned a criticality level, L, and is made up of a finite set of tasks. Each task, τ_i, is defined as $\tau_i = \{C_i, T_i, D_i, L_i\}$ where C_i is the computation time, T_i is the period (minimum inter-arrival time), D_i is the deadline and L_i is the criticality level. Each task gives rise to an unbounded series of jobs.

Vestal [81] observed an important relationship between the *Worst Case Execution Time* (WCET) estimation of a task and its criticality level. He observed that, as

the criticality level of a task increases so does its assigned WCET. Vestal reasons that this is due to increased pessimism involved in the analysis at higher criticality levels, resulting in larger WCET values. Burns and Baruah [28] note that it is also possible for a task to have criticality dependent minimum inter-arrival times (periods), the shorter the period, the higher the criticality level. This might be due to a task that is required to execute with increased frequency in a high criticality environment. This issue has been considered further in [14, 28, 10].

From this we can redefine our model to include vectors of values for both the WCET and the period, one for each criticality level, $\tau_i = \{\overrightarrow{C_i}, \overrightarrow{T_i}, D_i, L_i\}$ where $\overrightarrow{C}$ and $\overrightarrow{T}$ are vectors, one value for each criticality level. These vectors conform to the following, for any two criticality levels $L1$ & $L2$:

$$L1 > L2 \Longrightarrow C(L1) \geq C(L2)$$

$$L1 > L2 \Longrightarrow T(L1) \leq T(L2)$$

The possibility of a criticality dependent deadline has also been considered in [14]. Such a situation might entail a task having a shorter deadline for good quality of service and a longer safety critical deadline. This relationship would conform to the following:

$$L1 > L2 \Longrightarrow D(L1) \geq D(L2)$$

A system is able to utilise the model with its vectors of values, by adopting criticality change functionality. For a system with two criticality levels, L1 & L2 where L1$>$L2 (dual criticality) its behaviour would be as follows. The system would begin execution in the lower mode, L2, if any L1 criticality task executes to its L2 WCET budget without signalling completion, a criticality change would occur and the system would move into the higher criticality mode, L1. What happens to the L2 tasks when the system is executing in the L1 mode varies between approaches, but L1 tasks are allowed to execute up to their L1 WCETs, often at the expense of L2 tasks.

It is worth noting that Vestal's approach is based upon the assumption that the system designer's predictions for the lowest criticality level will be correct and that the system executing beyond these bounds is highly unlikely. The approach is a

means of providing the guarantees required by certification authorities while maintaining good system utilisation. As such only a rise in criticality level is considered.

We note a number of key definitions:

- **Criticality Level of a task:** This refers to, the criticality level of a particular task, e.g. X is a HI criticality task.
- **Criticality Level of the system:** This refers to the current criticality level (or mode) that the system is executing in, e.g. the system is executing in the LO criticality mode. In general during execution at criticality level (mode) L_i it is assumed that only tasks of whose criticality level is greater than or equal to L_i are provided guarantees that they will complete by their deadlines.
- **Assurance:** This is the degree of rigour require to satisfy a certification authority that a task will complete within the WCET provided at a particular criticality level. In general higher levels of rigour/assurance are required for higher criticality tasks.

2.1.1 An alternative approach to multiple WCETs

In the main portion of this chapter we described how each criticality level is associated with a WCET of its own. However, throughout this thesis we assume a slightly different model. While we permit multiple criticality levels, a task may have only 2 WCET values. One value at its own criticality level $C_i(L_i)$ and one at the lowest level in the system $C_i(LO)$ [29]. We define only two WCET estimates regardless of the number of distinct criticality levels as we view it as unlikely that a task will have the verification techniques of each level applied to it. Rather it need only have the techniques used for its own criticality level and those used for the lowest criticality level. Further mention and support for this model can be found in [27] and [64].

With the dual WCET model we describe the run-time and mode change functionality of a generic system. Each task has two WCET estimates, one for the lowest level in the system $C_i(LO)$ and one at its own criticality level $C_i(L_i)$. At any time, a task, τ_i, is in one of three possible states.

- The criticality level of the system is less than its own, thus it is executing using its $C_i(LO)$ WCET value.

- The criticality level of the system is equal to its own, thus it is executing using its own WCET value $C_i(L_i)$.
- The criticality level of the system is greater than its own, thus the task is not guaranteed to execute.

For example, consider a set of tasks spread across 3 criticality levels, level L_1 being the highest and level L_3 the lowest. The system begins execution at level L_3, if any level L_2 orL_1 task exceeds its L_3 WCET estimation the system moves into executing at criticality level L_2. Level L_2 tasks now execute at the estimation based on their own criticality level, level L_1 tasks continue to execute using the L_3 estimation. If any level L_1 task exceeds its L_3 WCET estimation the system moves into the highest criticality mode, all work apart from level L_1 tasks are suspended, level L_1 tasks may execute to their maximum WCET obtained using verification techniques required at their own criticality level.

Further discussion on the run-time of our cyclic executive mixed criticality model can be found in Chapter 3.

2.2 Uni-processor Scheduling

This section will describe the fundamental advances in mixed criticality uni-processor scheduling. We begin with fixed priority work initially instigated by Vestal [81] and move later to discuss Earliest Deadline First (EDF) and other policies.

2.2.1 Fixed Priority

The use of fixed priority scheduling in mixed criticality work began with Vestal's seminal 2007 work [81]. He defines the model below:

$$\tau_i = \{\overrightarrow{C_i(L_i)}, T_i, D_i, L_i)\} \tag{2.1}$$

Where T_i is the period, D_i is the deadline, L_i is the criticality level and $\overrightarrow{C_i(L_i)}$ is a vector of WCET values, one for each criticality level in the system. Vestal proposes a simple schedulability test based on standard Fixed Priority analysis, this test is given in equation (2.2):

$$R_i = C_i(L_i) + \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j(L_i) \tag{2.2}$$

This equation aims to calculate the response time, R_i, of task τ_i. If $R_i \leq D_i$ the task is schedulable. This equation uses $\tau_i \in hp(\tau_i)$ to refer to all tasks (τ_j) with a higher priority than τ_i. This equation is solved in a recursive fashion, replacing the resulting value of R_i back into the equation and re-calculating the result. When two successive values of R_i are equal iteration ceases and the response time of the task has been found. In this calculation the interference included for each higher priority task must use WCET values at the current tasks criticality level. As such, for a LO criticality task to have a higher priority than a HI criticality task it must be verified to the same level of assurance.

A fundamental evolution in both the system model and Fixed Priority analysis came in 2011 with a publication by Baruah et al. [15]. This evolution focused on the definition of two new FP scheduling schemes.

Static Mixed Criticality

Static Mixed Criticality (SMC) is a scheduling policy designed, in principle, to tackle the problem of criticality inversion. During the response time analysis proposed by Vestal [81] shown in equation (2.2) interference is considered from all tasks of a higher priority. Such tasks might have one of 3 relations depending on their criticality level.

1. If the criticality levels are equal, eg. $L_i = L_j$. In this case the value $C_j(L_j)$ may be used as the level of assurance is the same.

2. If the criticality level of the interfering task is greater, $L_i < L_j$. In this case, the value $C_j(L_j)$ is unnecessary, rather a value at the level of assurance L_i should be used, $C_j(L_i)$.

3. If the criticality level of the interfering task is less than the current task, $L_i > L_j$. Previously, in this situation the lower criticality task must also have a WCET at the same level of assurance as the task in question, giving $C_j(L_i)$. However verifying tasks to an unnecessary level of criticality is prohibitively

expensive. As such the WCET of a task is given at its own criticality level, $C_j(L_j)$, under promise of some assurance at run-time that this value is not exceeded.

Fundamentally, SMC introduces two new concepts. Firstly, each task need only be verified to its own level of assurance (criticality level) and those below. If during response time analysis the task in question is of a lower level of criticality the WCET used for the interfering task may be of that criticality level rather than its own. Secondly, some simplistic run-time monitoring is advantageous as it provides a guarantee that no task will exceed its WCET at its own criticality level.

SMC has two forms of response time analysis:

- SMC-NO: The first is SMC-NO, or Static Mixed Criticality with No Run-time Monitoring. SMC-NO is equivalent to Vestal's original algorithm and is shown in equation (2.2).
- SMC: The new Static Mixed Criticality algorithm utilises run-time monitoring to ensure that tasks need only be verified up to their own level of assurance (criticality level). The response time equation is shown in equation (2.3).

$$R_i = C_i(L_i) + \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j(min(L_i, L_j)) \tag{2.3}$$

Equation (2.3) uses $min(L_i, L_j)$ to indicate that if the τ_i has a lower criticality than τ_j the value $C_j(L_i)$ should be used, but if it has a greater criticality level $C_j(L_j)$ should be used.

Adaptive Mixed Criticality

Adaptive Mixed Criticality (AMC) built upon the notion of using run-time monitoring in order to better utilise platform resources. For a high criticality task, the WCET provided is often the result of very stringent analysis and, as such, is typically very pessimistic. Such pessimism is often driven by safety standards which result in margins of error being added to the predicted WCETs. While the WCETs at the highest level are very pessimistic, the execution times may be much lower in

practice. AMC takes advantage of this pessimism by allowing high criticality tasks to use the WCETs provided by assurance methods at lower levels of criticality. This is permitted on the assumption that, if the WCET at the lower criticality levels is exceeded, the task may still be given enough execution time to allow it to execute to its WCET at its own high criticality level. This is perhaps best illustrated by considering the runtime of AMC where there are two criticality levels, LO and HI (where HI > LO and thus $C(HI) \geq C(LO)$:

- The system begins execution in the LO criticality mode.
- If all tasks execute within their $C(LO)$ execution times, the system remains in the LO mode.
- If a HI criticality task executes up to its $C(LO)$ WCET without signalling that it has completed, a criticality change occurs from LO to HI.
- With the system in the HI criticality mode, LO criticality work is suspended (although LO criticality work currently pre-empted is allowed to complete) and HI criticality work is permitted to execute up to its maximum HI WCET ($C(HI)$.

Two schedulability tests were proposed for AMC [15], however, both share the same general principles as to which parts of the execution must be analysed. A three phase approach is required when considering a dual criticality system (HI / LO):

- Assess the schedulability of the LO criticality mode during which all tasks execute.
- Assess the schedulability of the HI criticality mode with only the HI criticality tasks executing.
- Assess the schedulability of the criticality change.

We begin by describing the first method, AMCrtb [15].

AMC-rtb

Adaptive Mixed Criticality - response time bound (AMCrtb) is a schedulability test for AMC systems. As stated above, three phases must be assessed. Phases one and two are stable (no mode change) and thus use analysis similar to that seen in equations (2.2) & (2.3):

Phase One: Check the schedulability tasks in the LO criticality mode.

$$R_i(LO) = C_i(LO) + \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{R_i(LO)}{T_j} \right\rceil C_j(LO) \tag{2.4}$$

Where hp is the set of higher priority tasks.

Phase Two: Check the schedulability of tasks in the HI criticality mode.

$$R_i(HI) = C_i(HI) + \sum_{\tau_j \in hpH(\tau_i)} \left\lceil \frac{R_i(HI)}{T_j} \right\rceil C_j(HI) \tag{2.5}$$

Where hpH is the set of higher priority, higher criticality tasks.

Phase Three: Finally, phase three must establish the schedulability of the criticality change itself, the response time ($R_i(LO)$ calculated in equation (2.4) is required.

$$R_i^*(HI) = C_i(HI) + \sum_{\tau_j \in hpH(\tau_i)} \left\lceil \frac{R_i^*(HI)}{T_j} \right\rceil C_j(HI) + \sum_{\tau_k \in hpL(\tau_i)} \left\lceil \frac{R_i(LO)}{T_k} \right\rceil C_k(LO) \tag{2.6}$$

Where hpH is the set of higher priority higher criticality tasks and hpL is the set of higher priority lower criticality tasks. The static value $R_i(LO)$ calculated earlier in equation (2.4) is used to bound the potential interference from LO criticality tasks during a criticality change as after the criticality change, all LO criticality tasks are suspended.

AMCmax

Adaptive Mixed Criticality - max (AMCmax) is an alternative approach for testing AMC schedulability which slightly outperforms AMCrtb at the cost of efficiency. The

premise of AMCmax is based on the idea that the criticality change can only occur at a finite number of points. In our model these points are when a task finishes its execution at the LO criticality level without signalling completion. AMCmax considers each of these points, searching for the worst case point of criticality change.

Given an allocation of tasks one may identify each completion time of a task in the low criticality mode. A point where a criticality change could occur is noted as time s. Given that there are a finite number of points of s, the number of points considered as possible change locations may be reduced to some time between time 0 and the LO response time of the given task ($R_i(LO)$). Figure 2.1 illustrates the points of s showing which points we must consider.

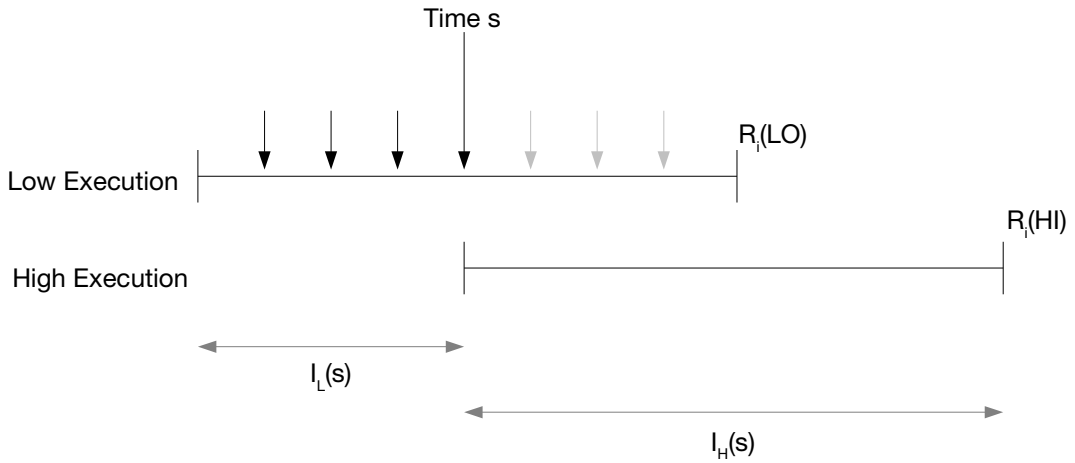


Figure 2.1: An illustration of the workings of AMCmax.

Figure 2.1 also illustrates the two segments of interference which must be accounted for, once a point of s has been selected. $I_L(s)$ is the interference during the LO criticality mode execution and $I_H(s)$ is the interference during the HI criticality mode execution. As such the response time of a given HI criticality task is given by equation (2.7):

$$R_i^s(HI) = C_i(HI) + I_L(s) + I_H(s) \tag{2.7}$$

Firstly, the low interference given in $I_L(s)$ is calculated using the algorithm shown in equation (2.8).

$$I_L(s) = \sum_{j \in hpL(i)} \left(\left\lfloor \frac{s}{T_j} \right\rfloor + 1 \right) C_j(LO) \tag{2.8}$$

This uses standard response time analysis techniques, differing slightly by making use of floor + 1 rather than ceiling. Floor + 1 is used to ensure that all tasks are accounted for, even those just released. This algorithm is also used for all tasks to assess their LO criticality response times, in this case $R_i(LO)$ is substituted for s and $I_L(s)$.

Calculating the HI criticality interference $I_H(s)$ is more involved. The HI criticality execution is considered as an interval of length ts (where $t > s$). t represents the response time of the task and is the value updated over each iteration of the algorithm. They begin by calculating the number of releases during the interval ts.

$$\left\lceil \frac{t - s - (T_k - D_k)}{T_k} \right\rceil + 1$$

A function M is defined to calculate the HI criticality interference. Three input parameters are given, k - the task, s - time and t - the response time. The function is shown in equation (2.9):

$$M(k, s, t) = min\left\{ \left\lceil \frac{t - s - (T_k - D_k)}{T_k} \right\rceil + 1, \left\lceil \frac{t}{T_k} \right\rceil \right\} \tag{2.9}$$

Ceiling + 1 is used to ensure all tasks are accounted for, despite a slight increase in pessimism. Given the value returned by M the LO criticality interference can be established.

$$\left(\left\lceil \frac{t}{T_k} \right\rceil - M(k, s, t) \right) C_k(LO)$$

As such $I_H(s)$ is given in equation (2.10):

$$I_H(s) = \sum_{k \in hpH(i)} \left\{ (M(k, s, t) C_k(HI)) + \left(\left(\left\lceil \frac{t}{T_k} \right\rceil - M(k, s, t) \right) C_k(LO) \right) \right\} \tag{2.10}$$

Finally, they declare the full calculation calculating both LO and HI criticality

interference, shown in equation (2.11):

$$R_i^s = \sum_{j \in hpL(i)} \left(\left\lfloor \frac{s}{T_j} \right\rfloor + 1 \right) C_j(LO) + \sum_{k \in hpH(i)} \left\{ (M(k,s,R_i^s) C_k(HI)) + \left(\left(\left\lceil \frac{R_i^s}{T_k} \right\rceil - M(k,s,R_i^s) \right) C_k(LO) \right) \right\} \quad (2.11)$$

With:

$$R_i = max(R_i^s) \forall_s$$

It is clear that AMC-max is a much more intensive schedulability solution and is considered intractable due to scalability issues.

AMC - extension to multiple criticality levels

AMC has been extended to include greater than two levels of criticality by Fleming and Burns [38][4]. They extend both AMC techniques, AMCrtb and AMCmax as well as adapting several other well known prior approaches such as, SMC and Period Transformation (PT). Below we show, in detail, the extension for AMCrtb and briefly cover the extension to AMCmax.

Where the analysis for AMCrtb (described above) for a dual criticality system is a three stage process, the extension on n criticality levels can be defined in two stages.

- *Stage 1*: The test must consider whether the task set is schedulable when executing at each criticality level in the system. In a system with 5 criticality levels, where $L1 > L5$, each criticality level must be checked. It is possible to illustrate this by showing the complete process for calculating the interference suffered by an $L5$ task, this is shown in equation (2.12).

[4]This work is referenced as while it was undertaken by the author of this thesis, it was included as part of a prior degree.

$$
\begin{aligned}
R_i(L5) = C_i(L5) + & \sum_{j \in hp(i) | L_j = L4} \left\lceil \frac{R_i(L5)}{T_j} \right\rceil C_j(L5) \quad + \\
& \sum_{k \in hp(i) | L_k = L3} \left\lceil \frac{R_i(L5)}{T_k} \right\rceil C_k(L5) \quad + \\
& \sum_{l \in hp(i) | L_l = L2} \left\lceil \frac{R_i(L5)}{T_l} \right\rceil C_l(L5) \quad + \\
& \sum_{m \in hp(i) | L_m = L1} \left\lceil \frac{R_i(L5)}{T_m} \right\rceil C_m(L5)
\end{aligned}
\tag{2.12}
$$

The interference of all higher priority tasks is considered with interfering tasks executing to their $L5$ execution times. This test is then repeated for each criticality level. Fleming and Burns [38] generalise this equation to account for n possible criticality levels.

For each criticality level:

$$\forall L \in 1 \ldots n$$

For all tasks where the criticality level is greater than or equal to L:

$$\forall \tau_i | L_i \geq L$$

Calculate the response times for that level.

$$R_i(L) = C_i(L) + \sum_{j \in hp(i) | L_j \geq L} \left\lceil \frac{R_i(L)}{T_j} \right\rceil C_j(L) \tag{2.13}$$

Equation (2.13) shows the generalised equation considering the interference suffered by τ_i at criticality level L. This test is carried out for each criticality level. If any level fails this test, the system is considered unschedulable.

- *Stage 2*: This stage is equivalent to Stage 3 of the dual criticality analysis as it considers the criticality change. Two segments of interference are identified; interference from higher priority tasks with a criticality level greater than or equal to the current level and interference from higher priority but lower criticality tasks. The first segment of interference can be calculated in the same way as each criticality level was calculated. The latter form of interference will have a bounded impact

on the overall response time of the task in question due to AMC suspending lower criticality tasks when a criticality change occurs.

Fleming and Burns [38] show the effect of higher priority but lower criticality tasks. They consider the effect on τ_i from a lower criticality but higher priority task τ_k.

$$\sum_{k \in hp(i) | L_k < L_i} \left\lceil \frac{R_i(L_k)}{T_k} \right\rceil C_k(L_k)$$

A static value, τ_i's response time at criticality level L_k, is used to bound the possible interference caused by τ_k before it is dropped due to the criticality change.

Fleming and Burns [38] then present the complete analysis for the interference suffered during a criticality change with n possible levels of criticality, this is shown in equation (2.14).

For each criticality level:

$$\forall L \in 1 \ldots n$$

For all tasks where the criticality level is greater than or equal to L:

$$\forall \tau_i | L_i \geq L$$

Beginning at the lowest criticality level, calculate the schedulability of each criticality change.

$$\begin{aligned} R_i^*(L) = C_i(L) + & \sum_{j \in hp(i) | L_j \geq L} \left\lceil \frac{R_i^*(L)}{T_j} \right\rceil C_j(L) \quad + \\ & \sum_{k \in hp(i) | L_k < L_i} \left\lceil \frac{R_i(L_k)}{T_k} \right\rceil C_k(L_k) \end{aligned} \tag{2.14}$$

AMCmax is also extended to facilitate n possible criticality levels. The approach is conceptually fairly simple, for each of the original points (s_1) that a criticality change could occur, there are a number of additional points at which the next criticality change might take place. This is illustrated in Figure 2.2.

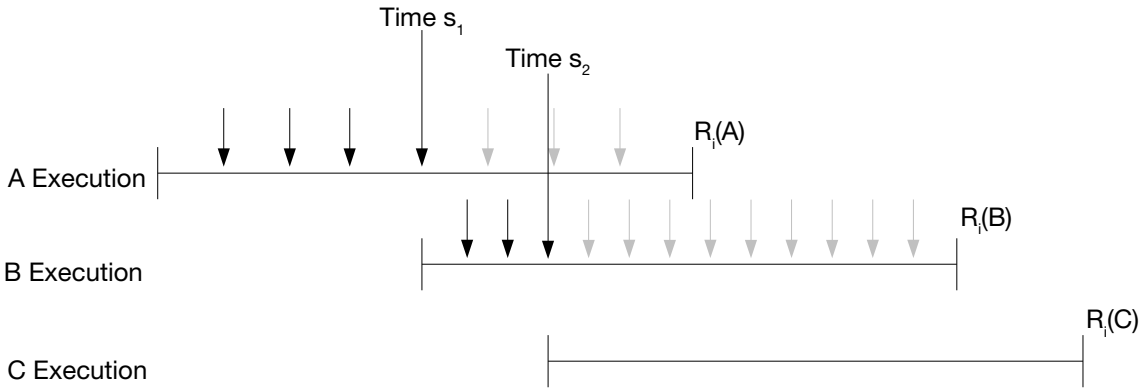


Figure 2.2: The system with modes A and B with an additional level, C added.

Figure 2.2 shows three criticality levels A, B and C (where $A < B < C$). Much like the example shown in Figure 2.1 this demonstrates the possible points that a criticality change might occur. When compared with Figure 2.1 it is possible to see how another set of points must be searched in order to determine the points of change that will cause the worst case response time. With the addition of these points it is also clear that the problem of searching each point becomes very computationally intensive as criticality levels are added. For each original point of change there might be a large number of points to check for subsequent changes. As such, Fleming and Burns [38] conclude that while AMCmax dominates AMCrtb, its improvement is not significant enough to warrant the intensity of its search.

Zhao et al. [83] extend AMC to incorporate preemption thresholds. Their results show PT-AMC provides a slight improvement over AMC-rtb and AMC-max. Baruah and Chattopadhyay [16] extended AMC to account for varying periods. Burns and Davis [30] also consider criticality dependent frequency.

AMC-IA vs AMCmax

Huang et al. [47] aim to provide a comparison between many different mixed criticality scheduling approaches for the sporadic task model. As part of this work they examine the scheduling policies proposed by Baruah et al. [15], AMCrtb and AMCmax. In addition they propose their own approach, AMC-IA which claims to outperform both previous approaches. They show that their algorithm is extendable to n possible criticality levels. In the discussion below we consider a comparison of AMCmax and AMC-IA based on their dual criticality analysis.

The original AMCmax algorithm is described below. The technique works on the premise that there are a finite number of points in which a criticality change could occur. These points, s, are bounded by $0 < s < R_i(LO)$. Figure (2.3) (also shown in 2.2.3, repeated here for convenience) shows a set of such points:

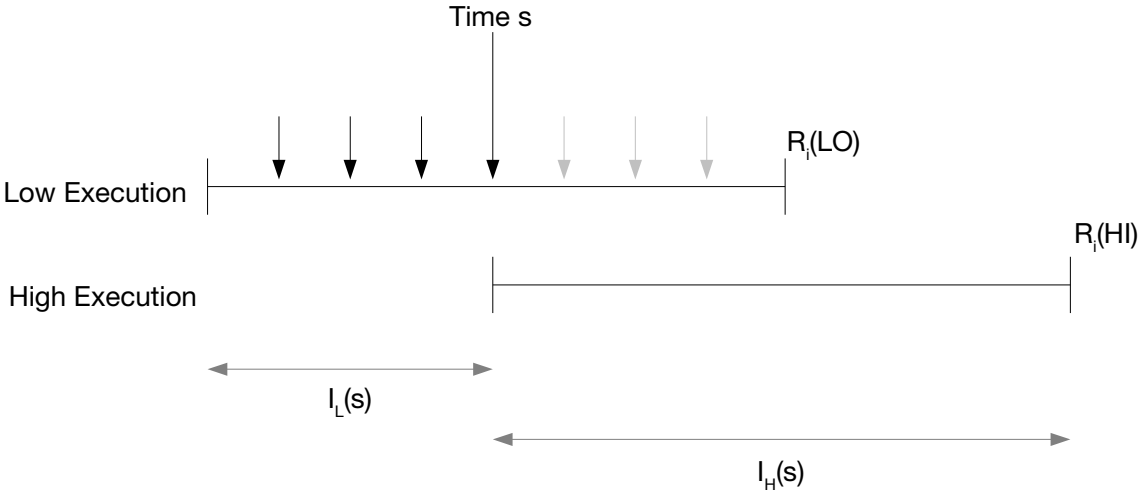


Figure 2.3: Example AMCmax criticality change.

Figure (2.3) also shows the two segments of interference we are interested in, $I_L(s)$ and $I_H(s)$. $I_L(s)$ represents the interference suffered by all tasks in the LO mode, $I_H(s)$ represents the interference suffered by HI criticality tasks after the criticality change at time s. The full equation for AMCmax is shown in equation (2.15):

$$R_i^s = C_i(HI) + \sum_{j \in hpL(i)} \left(\left\lfloor \frac{s}{T_j} \right\rfloor + 1 \right) C_j(LO) + \sum_{k \in hpH(i)} \left\{ (M(k,s,R_i^s) C_k(HI)) + \left(\left(\left\lceil \frac{R_i^s}{T_k} \right\rceil - M(k,s,R_i^s) \right) C_k(LO) \right) \right\} \quad (2.15)$$

And:

$$R_i = max(R_i^s) \forall s$$

Where hpL considers the set of higher priority, LO criticality tasks and hpH considers the set of higher priority, HI criticality tasks. Function M is used to calculate

the interference suffered by a HI criticality task in the HI criticality mode.

$$M(k,s,t) = min\left\{\left\lceil \frac{t-s-(T_k-D_k)}{T_k} \right\rceil + 1, \left\lceil \frac{t}{T_k} \right\rceil\right\} \tag{2.16}$$

Finally, Baruah et al. [15] note that a point of s may be only on the release of a LO criticality task.

Haung et al.'s [47] approach[5] uses the same notion of a number of points of s at which a criticality change could occur. They define two methods of calculating the LO criticality interference. One for the interference suffered from all higher priority tasks in the LO mode, shown in equation (2.17):

$$n_j(s) = \left\lceil \frac{s}{T_j} \right\rceil \tag{2.17}$$

And one to calculate the interference of the higher priority, HI criticality tasks during their LO criticality execution, shown in equation (2.18):

$$n_k(s) = max\left(\left\lfloor \frac{s-D_k}{T_k} \right\rfloor + 1, 0\right) \tag{2.18}$$

It should be noted that both equations appear to be the same function, n, they are different functions differentiated by subscript j and k.

The completed algorithm for AMC-IA can be seen in equation (2.19):

$$R_i^s = C_i(LO) + \sum_{\tau_j \in hp(i)} n_j(s)C_j(LO) + \sum_{\tau_j \in hpH(i)} \left(\left\lceil \frac{R_i^s}{T_k} \right\rceil - n_k(s)\right)C_k(HI) \tag{2.19}$$

And:

$$R_i^* = max\left\{R_i^* | \forall_{\tau_j}, s \in \left[D_j, ..., \left\lceil \frac{R_i(LO)}{T_j} \right\rceil T_j + D_j\right]\right\}$$

There are several issues with the presentation of this algorithm. Firstly, it seems clear that as equation(2.19) is calculating the response time of a HI criticality task, $C_i(HI)+$ should be used rather than $C_i(LO)+$. Secondly, the use of the letter n to represent two functions causes some confusion, we might re-write equation (2.18)

[5]We alter their notation to be comparable with AMCmax.

as a function m:

$$m_k(s) = max\left(\left\lfloor \frac{s - D_k}{T_k} \right\rfloor + 1, 0\right) \tag{2.20}$$

And therefore the complete equation:

$$\begin{aligned} R_i^s = C_i(HI) + \sum_{j \in hpL(i)} n_j(s) C_j(LO) + \\ \sum_{k \in hpH(i)} m_k(s) C_k(LO)) + \\ \left(\left(\left\lceil \frac{R_i^s}{T_k} \right\rceil - m_k(s)\right) C_k(HI)\right) \end{aligned} \tag{2.21}$$

The key difference between these two algorithms is the way in which they calculate the areas of interference in the LO and HI mode. AMCmax uses the function M, as seen in equation (2.16), in order to determine the interference suffered in the HI criticality mode by higher priority HI criticality tasks. The LO criticality interference for these HI tasks is calculated by removing the value of M from the total:

$$\left\lceil \frac{R_i^s}{T_k} \right\rceil - M(k, s, R_i^s)$$

The interference suffered by higher priority, LO criticality tasks for AMC-MAX is calculated using:

$$\left\lfloor \frac{s}{T_k} \right\rfloor + 1$$

AMC-IA differs by using the first segment of the algorithm to calculate the LO interference of all tasks, this is done using the functions $n_j(s)$ and $m_k(s)$ seen in Equations (2.17, 2.19). The approach then uses the value derived from $m_k(s)$ for the HI criticality, higher priority tasks to calculate their HI mode interference by removing it from the total number of releases:

$$\left\lceil \frac{R_i^s}{T_k} \right\rceil - m_k(s)$$

Essentially, AMCmax calculates the HI interference and removes it from the

total find the LO interference and AMC-IA calculates the LO interference and removes it from the total to find the HI interference.

The majority of the time, AMCmax and AMC-IA will produce the same results. However, there is a scenario where AMC-IA will schedule a task set which is, in reality unschedulable. Such a situation might occur when all tasks are not released at time 0, thus causing a greater level of HI criticality interference than AMC-IA accounts for.

We have found an ordering of tasks which, although claimed schedulable by AMC-IA, is in reality not feasible[6]. Consider the task set shown in Table (2.1):

	C(LO)	C(HI)	T/D	L	P
τ_1	2	-	10	LO	1
τ_2	4	6	10	HI	2
τ_3	10	15	40	HI	3

Table 2.1: AMC-IA, problematic example.

The LO response time of τ_3 can be calculated:

$$R_3(LO) = 10 + \left\lceil \frac{28}{10} \right\rceil 2 + \left\lceil \frac{28}{10} \right\rceil 2$$

Giving the result of 28.

Based on the assumption made by Haung et al. [47] that points of s must correspond to task deadlines, there are only two possible points of s at time 10 and at time 20. $s = 10$ results in the worst case response time for AMC-IA and AMCmax; we will now consider the calculation using both algorithms.

The calculation for AMC-IA is as follows:

$$R_3^{10} = 15 + \left\lceil \frac{10}{10} \right\rceil 2 + \left(\left\lfloor \frac{10-10}{10} \right\rfloor + 1 \right) 4 + \left(\left\lceil \frac{39}{10} \right\rceil - \left(\left\lfloor \frac{10-10}{10} \right\rfloor + 1 \right) \right) 6$$

A result of 39 (as $39 < 40$) shows that AMC-IA claims to be able to schedule τ_3 at the lowest priority.

We now show the calculation for AMCmax:

[6]This example was produced by the author of the thesis as part of the corrections to AMC-IA published in [39].

$$R_3^{10} = 15 + \left(\left\lfloor \frac{10}{10} \right\rfloor + 1\right)2 + \left(\left\lceil \frac{49-10}{10} \right\rceil + 1\right)6 + \left(\left\lceil \frac{49}{10} \right\rceil - \left(\left\lceil \frac{49-10}{10} \right\rceil + 1\right)\right)4$$

With a results of 49, AMCmax clearly shows τ_3 as being unschedulable at priority 3.

In this situation AMC-IA does not account for all possible scenarios. Consider the example shown in Figure (2.4):

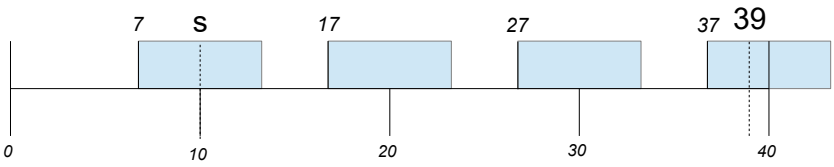


Figure 2.4: Example execution with τ_2 release at time 7.

If τ_2 was released at time 7, its execution would continue into the HI mode. τ_2 could execute again at times 17, 27 and 37, each constituting an execution of $C_2(HI)$. However AMC-IA accounts for only 3 possible HI executions and as such is not a sufficient test in this situation. This flaw in AMC-IA accounts for the small number of task sets it claims to be able to schedule that AMCmax cannot.

2.2.2 EDF

The first work to consider EDF for mixed criticality was completed in 2008 by Baruah and Vestal [21]. In this work they note that EDF in its current form is not optimal for mixed criticality systems, unlike normal (non mixed criticality) systems. They also extend Vestal's original model [81] to include sporadic tasks.

Park and Kim [66] develop a technique known as Criticality Based Earliest Deadline First (CBEDF). CBEDF utilises slack in the system and dynamically reallocates this to LO criticality tasks. This includes slack generated by tasks completing early (remaining slack) and any remaining execution time if all HI tasks are executing to their allocated time (empty slack). CBEDF is then compared to Own Criticality Based Priorities (OCBP), a scheduling approach proposed in [20], their experimental results show that CBEDF dominates OCBP.

Baruah et al. [13, 12] propose an EDF technique known as Earliest Deadline First with Virtual Deadlines (EDF-VD). EDF-VD (in a dual criticality system) allows for HI criticality deadlines to be artificially reduced during system execution in the LO mode. Deadlines are reduced to ensure that, if a criticality change occurs, HI criticality tasks would have time remaining to complete before their deadline. If a criticality change occurs, LO criticality tasks are dropped. EDF-VD is extended for multi-processor systems in [60, 17].

Ekberg and Yi [35] build upon the work undertaken by Baruah et al. [13] on EDF-VD. They propose the use of two relative deadlines for a task, one being the actual deadline and another is an earlier artificial deadline which may be tuned to allow HI criticality tasks to execute before LO. They utilise demand bound functions to tune these artificial deadlines.Their experimental work compares well known approaches such as Vestal's algorithm [81], OCBP [20] and EDF-VD. Their results show a performance improvement over previous approaches, crucially this represents a case where an EDF based approach out-performs previous fixed priority work such as AMCmax [15] or OCBP. Their work is extended in [36] to allow all task parameters to vary with the criticality level. Easwaran [34] presents a new demand bound function shown to strictly dominate those in [35].

Huang et al. [48] note that simply dropping LO criticality tasks when a criticality change occurs is not acceptable. Their approach builds from EDF-VD but looks to guarantee a level of degraded service to LO criticality tasks when a criticality change occurs. Their work is based around providing functionality for an industrial Flight Management System, as part of this they suggest an approach which allows the system to be reset and provide analysis to quantify the time this would take.

2.2.3 Period Transformation

Vestal [81] suggests that an adapted form of period transformation could be applied to mixed criticality task sets. Rather than aiming to create harmonic task sets, the aim is to allow for a criticality monotonic priority ordering, thus avoiding the problem of criticality inversion[7]. As the aim is to ensure a criticality monotonic ordering

[7] A high priority LO criticality task can preempt a low priority HI criticality task potentially causing a deadline miss.

only certain tasks must be transformed. LO criticality tasks do not need to be transformed, HI criticality tasks with a period shorter than the shortest LO period need not be transformed, HI criticality tasks with a period greater than the shortest LO period must be transformed.

Fleming and Burns [38] consider Vestal's technique and complete the analysis. The analysis for HI criticality tasks running in the HI mode is done via standard response time techniques [4]. For the LO mode, HI criticality tasks requiring transformation are transformed by a factor, m:

$$m = \left\lceil \frac{T_j}{T_l} \right\rceil$$

Where τ_l is the LO criticality task with the shortest period and τ_j is a HI criticality task that must be transformed.

The runtime behaviour of a transformed mixed criticality system differs slightly from the norm. The system begins execution in the LO criticality mode, transformed tasks execute for their $C_j(HI)/m$ until they execute up to their untransformed $C_j(LO)$. Only at this point can we say that a task has overrun its budget and a criticality change must occur.

It is possible to calculate the number of transformed executions of τ_j that might interfere with τ_i.

$$\left\lceil \frac{R_i}{T_j/m} \right\rceil$$

The value produced from the above calculation will include several complete executions of $C_j(LO)$ and some remaining transformed executions, $C_j(HI)/m$ which do not make up a complete execution of $C_j(LO)$. Those values which constitute a complete $C_j(LO)$ can be calculated as shown:

$$\left\lfloor \frac{R_i}{T_j} \right\rfloor C_j(LO)$$

Vestal [81] introduces additional pessimism by stating that the remaining transformed executions must be less than $C_j(LO)$, as such he assumes this value as the worst case.

$$\left\lfloor \frac{R_i}{T_j} \right\rfloor C_j(LO) + C_j(LO)$$

In many cases this may be overly pessimistic. Fleming and Burns [38] define a way of calculating the interference suffered from the spare transformed executions. The size of the remaining interval, P, can be calculated:

$$P = R_i - \left\lfloor \frac{R_i}{T_j} \right\rfloor T_j$$

We can use this interval to determine the number of transformed executions, x:

$$x = \left\lceil \frac{P}{T_j/m} \right\rceil \frac{C_j(HI)}{m}$$

This can be included in the equation which also considers complete executions of $C_j(LO)$, see equation (2.22):

$$min\{x, C_j(LO)\} + \left\lfloor \frac{R_i}{T_j} \right\rfloor C_j(LO) \tag{2.22}$$

The use of min ensure that there is no potential for undue pessimism from the calculation of the untransformed tasks.

Although not all tasks are transformed, period transformation still suffers from the problem of high overheads. Fleming and Burns [38] state that, although it is possible to adapt period transformation to schedule mixed criticality systems, its performance would be poor if its high overheads were accounted for.

2.3 Multi-Processor Scheduling

In 2009 Anderson et al. [3] undertook the initial work into mixed criticality systems on multi-core platforms. As with historical multi-processor/core work [33], mixed criticality implementations face the same challenges. Current research has focused upon two key approaches, either Partitioned in which tasks are statically assigned to a processor, or Global where tasks can migrate at run time depending on demand and available slack. Some work provides example frameworks while others provide more analytical scheduling approaches.

2.3.1 Partitioned

Partitioned multi-core systems use servers (containers) to schedule applications of different criticality levels while maintaining the isolation required and potentially allowing servers to migrate from one core to another. There are two key topics in partitioned scheduling, task allocation and scheduling approaches within each partition. Task allocation is the process of assigning tasks to different partitions or processors. Once tasks or partitions are assigned a processor/s, it is important to consider how such tasks might be scheduled.

Bastoni et al. [23] present a comparison between global, partitioned and clustered EDF scheduling. Their results show that G-EDF (Global EDF) is never preferable for Hard Real Time Tasks with P-EDF (Partitioned EDF) performing the best. C-EDF (Clustered EDF) is a technique that aims to provide a mid ground between a fully global scheduling approach and a fully partitioned one, this involves clustering the platform into groups of cores which share a cache. A key aspect of this investigation is that, somewhat unusually, run time overheads were explicitly considered.

Lakshmanan et al. [55] address the need for a bin packing scheme in partitioned multi-core systems by suggesting their, Compress-on-Overload Packing (COP) scheme. The scheme aims to increase high-criticality tasks' tolerance to overload conditions. They examine the effectiveness of their scheme using a metric to determine a system's reliance to overload. A system is considered highly ductile if HI criticality tasks are able to continue to meet their deadline at the expense of LO criticality execution. COP is compared to another allocation approach, Worst-Fit Decreasing (WFD) and show that COP dominates WFD.

Kelly et al. [53] present a comparison between two priority assignment techniques, Rate Monotonic (RM) and Audsley's Algorithm [6], used in conjunction with two partitioning schemes, Decreasing Utilisation (DU) and Decreasing Criticality (DC). They then compare DC-Audsley, DU-Audsley, DC-R and DU-RM. They show that Audsley's algorithm out performs RM and that DC out performs DU, but suffers as the number of tasks per CPU is increased.

Tamas-Selican and Pop [72, 73] consider the issue of mapping mixed criticality applications on distributed platforms. In order to do this they use a simulated

annealing based approach to optimise task mapping using time-partitions.

2.3.2 Global

Li and Baruah [60] extend EDF-VD [11] to multiprocessor systems. This is done by applying a known global scheduling scheme, fpEDF [9] to EDF-VD. In 2013 Baruah et al. [17] consider EDF-VD for multi-core and explore global and partitioned approaches. They conclude, through experimental results and observation that partitioned approaches are more practical, although advances in global scheduling could be made.

Pathan et al. [67] present a Global scheme known as Mixed-criticality Scheduling on Multiprocessors (MSM). This algorithm is based upon previously uni-processor techniques such as AMC. They compare their technique with priorities assigned via Audsley's algorithm [6], deadline monotonic and criticality monotonic orderings. They show that Audsley's algorithm outperforms the other two approaches and claim extendability to more than two criticality levels.

Saraswat et al. [69] present a Tabu Search [43] approach, utilising EDF scheduling and the Constant Bandwidth Server (CBS) technique [1] to enforce temporal isolation where required.

2.3.3 Other Approaches

Mollison et al. [63] extend the multi-processor scheduling platform known as MC^2, originally presented in [3]. The technique uses both global and partitioned approaches depending on the criticality level of the application in question. Higher criticality levels are assigned to a processor (partitioned) and run under cyclic executive or EDF policies, such tasks may not migrate to another core. Lower criticality applications are scheduled via a global approach using either G-EDF (Global-EDF) or a best effort scheme, such tasks are allowed to migrate between cores at runtime.

Herman et al. [46] make use of MC^2 addressing implementation issues. They implement MC^2 on the linux based, $LITMUS^{RT}$ [26] platform and show that an implementation is possible while maintaining a suitable level of isolation for HI crit-

icality tasks and keeping system overheads relatively low.

2.4 Linear Programming

Linear Programming (LP) developed from the study of linear inequalities. Most modern applications began with Kantorovich [52] in 1939, but remained unknown and unnoticed. During and just after World War II Linear Programming became a popular tool for dealing with complex planning or scheduling problems. The process of defining such models is described as "(We shall use) the term model building to express the process of putting together symbols representing objects according to certain rules" [32]. The models produced aim to describe a scenario, its constraints and possible solutions, these building blocks "often result(s) in a system of linear inequalities and equations; when this is so, it is called a linear programming model". Linear Programming may be defined as "(...) the optimization of a linear function over a feasible set defined by linear inequalities" [77], in short, Linear Programming is the process of maximising or minimising some linear function based on a set of variables with a number of linear constraints. The Introduction of the Simplex Method in 1947 by Danzig [32] helped popularise LP. The Simplex Method allowed for Linear Programming formulations to be solved efficiently. There is a huge amount of work on Linear Programming and its solutions [80], however, this thesis focuses on its modern and practical applications for mixed criticality systems.

Modern use of Linear Programming makes use of powerful solvers able to deliver solutions to complex problems in reasonable timeframes. Common commercial solvers include the likes of Gurobi [45] and CPLEX [49], while open source alternatives, such as LP_Solve [24], also exist. Typically these tools are treated in a black box fashion, a model is input and a result is output, little detail is gained about the exact process taken to reach the result. This is particularly true in the case of the proprietary commercial solvers and is true to a lesser extent in the case of open source alternatives. Commercial solvers tend to benefit from a greater investment toward speed and efficiency, while open source alternatives provide a more transparent tool.

Two common forms of Linear Programming are commonly used in a real-time mixed criticality context: Integer Linear Programming (ILP) and Mixed Linear Programming (MLP). An ILP model is a Linear Programming model defined using only integer variables. Similarly, a MLP model is a Linear Programming model defined using a mixture of integer and continuous variables. The use of integer variables for real-time problems is often dictated by the notion of time as a discrete integer value. The most typical real-time mixed criticality application for Linear Programming (and its subsets ILP, MLP) is some form of allocation problem. Such a problem might be anything from priority assignment to resource allocation. Each allocation is typically performed with an optimisation goal in mind, some examples of allocations with a variety of optimisation goals are described below:

- Fault Tolerance: This might take the form of allocating CAN message priorities [7] which even accounts for a situation where parts of a message are given different priorities. The solution is optimised to reduce the total number of priorities required. An allocation problem may simply tackle task priority assignment [76]. Task re-execution is considered which may be required to occur on different processors. The goal is to find a suitable priority assignment while minimising the number of processors used. Linear Programming may even be utilised to find allocations which may account for permanent processor failures [2].

- Reduced Power Consumption: A common goal when optimising an allocation on a real-time platform is to reduce the power consumption. Work such as [58, 57, 56] tackles this issue using linear programming to allocate tasks in a power aware manner. These linear programs are optimised to reduce the static power consumption.

- Resource Access: Some optimisation problems seek to allocate work with a specific focus on resource access. In the case of [65] a multi-core allocation is considered where the allocation is optimised to reduce the Worst Case Resource Access.

- Security: Allocation problems may also feature security issues. Work in [82] uses Mixed Linear Programming to optimise the tradeoff of maximising band-

width utilisation while maintaining real-time and security guarantees on a CAN network. The approach uses a MAC based security system to defend CAN against attacks.

- Other: Other uses look to provide graceful degradation [54], schedule an Engine Control Unit (ECU) and bus toward the goal of system stability and overall performance[44], calculate stretching factors for LO criticality tasks [51], allocate criticality aware Last Level Caches [31] or simply construct scheduling tables [19].

2.5 Cyclic Executives

The Cyclic Executive scheduling policy [8] is a highly deterministic scheme which has been well adopted in industry. The CE schedule is made up of two key components:

- The Major Cycle: described as ”the sequence of actions to be performed during some fixed period of time“ [8]. The work executed in the major cycle is repeated upon completion of the cycle.

- The Minor Cycle: a number of minor cycles (also referred to as frames) make up a major cycle and are allocated work. Minor cycles are restricted such that “no frame may be longer than the shortest period (of a task) ” [8]. In addition, while it is not strictly required, common practice is to make all minor cycles of equal length.

All work is statically allocated offline to minor cycles. The static nature of the cyclic executive lends itself greatly to applications where detailed knowledge of the execution is desirable. Safety critical applications require stringent verification, which is eased significantly when the chosen scheduling policy is simple and highly deterministic.

The cyclic executive structure is illustrated in the introduction, see Figure 1.1. In this diagram, T^M is used to represent the length of the minor cycle, shown with a line across the top encompassing all execution. The notation T^F is used to indicate the minor cycle length, shown by each of the boxes that make up a major cycle.

While the determinism of a cyclic executive is valuable, they are restrictive in terms of the parameters of the work they can schedule. The key requirements on task parameters are as follows:

- Tasks must have periods that are multiples of the minor cycle.
- Tasks must have periods that are no greater than the major cycle.
- Tasks must have deadlines greater than or equal to the minor cycle.

As the periods of tasks and the minor cycle length are so closely coupled, the design of the cyclic executive scheduler must be performed alongside the development of tasks. Applications designed for cyclic executive execution are often designed with these restrictions in mind, the trade-off being a highly deterministic and predictable platform to aid certification.

To this point, the primary use of cyclic executives in the mixed criticality context is as a means of scheduling only the highest criticality tasks. The use of a cyclic executive policy for the highest criticality levels in a partitioned MC system was proposed in [3] and further expanded in [63]. In this case a cyclic executive policy is used for only the highest level of criticality due to its high determinism. Its relatively poor resource utilisation prohibits its use at other criticality levels. Implementation issues of the approach are investigated in [46] and optimisation and allocation issues are discussed in [31].

Other uses for cyclic executives in the context of mixed criticality include the use of CE scheduling by a hypervisor responsible for mixed criticality partitions [78].

2.5.1 Time Triggered

Time Triggered (TT) scheduling is, in some sense, the superset of cyclic executive scheduling. It requires tasks to execute at precomputed times as specified by a scheduling table. The first to consider the TT paradigm in the mixed criticality context were [18], their work focused on a TT set-up where multiple scheduling tables were specified, one for each criticality mode. The problem of generating permissible scheduling tables has been tackled using heuristic techniques in [75] and by using Linear Programming tools in [44][50]. The work of [70] and [71] consider how

Fixed Priority (FP) or Fixed Priority per Mode (FPM) can be transformed into suitable scheduling tables for a mixed criticality platform. This TT approach utilises the so called Single Time Table per Mode (STTM) paradigm. Finally, the incorporation of legacy scheduling tables into modern mixed criticality systems is addressed in [74].

2.6 Barrier Protocol

Barrier synchronisation is a fundamental concept to multi-processor computing. Initially proposed by Valiant [79] as part of the Bulk Synchronous Parallelism (BSP) paradigm which provides a mechanism for the synchronisation of parallel workloads on multi-processor platforms. BSP centred around the notion of supersteps, during each superstep a processor can perform computation and send and receive messages, all work and communication must complete within each superstep. At the end of the superstep barrier synchronisation is employed to check if all processors have completed their allocated work, if they have the next superstep begins execution, if they have not, the execution time that that would have been used by the next superstep is provided to allow all processors to complete execution.

The notion of utilising barrier synchronisation to ensure that processors are in a consistent state before beginning a new set of tasks is useful in a mixed criticality context. A big challenge when scheduling mixed criticality workloads is managing the required level of isolation between tasks of differing criticality levels. This extends beyond execution to include potential interference of resources. The work of Giannopoulou et al. [42, 41] addresses this problem by utilising a Time Triggered approach.

In their work, tasks are isolated such that only a statically known number of tasks may contest resources at any given time. To achieve this a barrier protocol is used. The barrier provides a point of system-wide synchronisation (across all cores) allowing the system to ensure only tasks of a particular criticality level are permitted to execute within a given time.

Consider the situation where a system contains 3 levels of criticality and 2 processing cores. Figure 2.5 illustrates how a potential schedule might look.

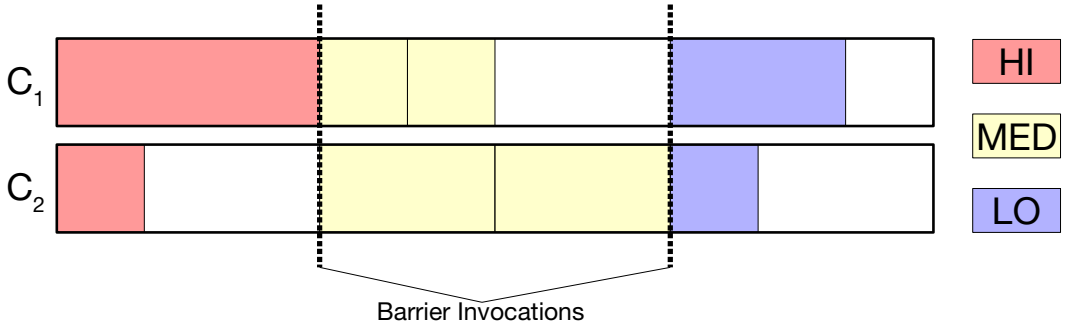


Figure 2.5: An example schedule illustrating the usage of a barrier protocol.

The coloured areas represent execution of a task with tasks being coloured red, yellow and blue to correspond to HI, MED and LO criticality modes respectively (HI>MED>LO). Each dashed vertical line indicates a point at which both cores simultaneously move to the next execution phase and allow the next criticality level to execute. The system begins executing the highest level and continues to the lowest with a barrier separating each level. The points illustrated in Figure 2.5 are found offline using the WCET of each task. However, at runtime the barrier is a dynamic protocol, the change from the execution of one criticality level to the next occurs when work has completed across all cores, thus providing good system utilisation. This approach aims to provide a level of dynamic behaviour which is acceptable for high criticality applications while providing improved system utilisation.

2.7 Summary

This chapter has provided an overview of the mixed criticality literature relevant to this thesis. The development of the mixed criticality task model was described and some early work on mixed criticality scheduling was covered. Mixed criticality literature that makes use of linear programming was discussed. Finally, work focusing on cyclic executive and time triggered platforms in a mixed criticality context was examined.

The field of mixed criticality was and still is developing rapidly. While a large amount of work has focused on more progressive and dynamic approaches to this difficult scheduling problem, it is clear from this review that a number of authors are still taking a more static approach. In this thesis we consider the design and allocation of real-time mixed criticality systems on multi-core platforms utilising cyclic

executive architecture. We aim to provide a balance between predictability, isolation, legacy support and good system utilisation.

Chapter 3

Task Allocation

This chapter deals with the fundamentals of finding feasible task allocations for mixed criticality cyclic executives on multi-core processors. The problem of constructing cyclic executive schedules is well known to be NP-hard [8] even for a single processor. The allocation of tasks to a cyclic executive is considered similar to the well known bin packing problem [40]. The schedulability of a standard (non-mixed criticality) cyclic executive can be determined simply by construction, if it is possible to create a schedule, then by definition that allocation is schedulable.

We choose a Cyclic Executive scheduler to remain in-line with industrial practice, Cyclic Executive schedulers are widely used in safety criticality systems. Industrial partners such as BAE Systems[1] influenced our decision to use a Cyclic Executive. Its use allowed our work to be more relevant and relatable to industrial partners.

In the mixed criticality case, we provide the isolation required by each criticality level by completely separating the execution, this is in-line with the state of the art [41, 42]. Tasks of the highest criticality level are always executed first, followed by each level below, with the system only ever executing tasks of any one level of criticality. From the perspective of a certification authority, a task at the highest criticality level need only be concerned with executing alongside other tasks of the same level (as any task that can affect the execution must also be verified to the same level of assurance). Any potential interference based on factors such as communication or shared resource access can only occur from another task at the

[1] http://www.baesystems.com

same level of criticality. All other levels execute after and are completely temporally separated, no lower criticality tasks can have impact on higher criticality tasks. As this is a mixed criticality system, we consider the expected execution to occur when all work executes to their WCETs calculated using the assurance of the lowest criticality level in the system. During normal execution the highest level will begin executing to the lowest assurance, if execution at the highest level fails to complete by its LO assurance prediction, we adopt the standard model which assumes all other work is dropped (or not guaranteed) and the highest criticality level is given time to execute to its most pessimistic WCET. We adopt the dual WCET model as described in Section 2.1.1, such that each task has only two WCET values, one verified to the required standard for its own criticality level and one verified to the standard of the lowest criticality level in the system.

In this chapter we focus on a dual criticality model for simplicity. A task's τ_i WCETs are $C_i(LO)$ & $C_i(HI)$ where $C_i(LO) \leq C_i(HI)$. The cause of such a system to move from the *standard* (LO) execution mode to the HI (where all LO work is not guaranteed) is when all HI criticality work does not complete by a pre-computed point. This point is constant across all cores in the system and is denoted by S^{max}. This distinction deviates from the more common model assumed by Vestal [81] and others which assumes that the criticality change occurs if any one particular task executes for greater than its $C(LO)$ execution time. S^{max} is the synchronised point of switching across all cores.

More formally, let $S(i,j)$ denote the latest instant at which a core i signals completion of HI work in minor cycle j. Let $S^{\max}(j)$ denote the instant at which LO work may commence across the entire system. Schedulability of a dual criticality cyclic executive can be calculated as follows (where $HI(i,j)$ and $LO(i,j)$ denote respectively the sets of HI-criticality and LO-criticality tasks scheduled on core i, minor cycle j):

1. Check the schedulability of HI criticality tasks:

$$\forall i \text{ and } j, \sum_{k \in HI(i,j)} C_k(HI) \leq T^F$$

2. The value of $S(i,j)$ can be calculated for each core:

$$S(i,j) \ = \ \sum_{k \in HI(i,j)} C_k(LO)$$

3. The value of $S^{\max}(j)$ used across all cores is:

$$S^{\max}(j) = \max(S(i,j))$$

4. LO criticality jobs must fit within the time between $S^{\max}(j)$ and the end of the minor cycle:

$$\forall i \text{ and } j, \sum_{k \in LO(i,j)} C_k(LO) \leq T^F - S^{\max}(j)$$

We may now describe the runtime of a dual criticality cyclic executive on a multi-core platform.

- HI criticality begins execution.
- If all HI criticality work completes by time $S^{\max}(j)$, LO criticality work begins execution.
- If all HI criticality work does not complete by time $S^{\max}(j)$, HI criticality tasks are given up to their $C(HI)$ execution times to complete[2].

The nature of the synchronised criticality switching point, $S^{\max}(j)$, may be considered in multiple ways. Our offline analysis treats this as a pre-computed point, at which an interrupt is fired either starting the execution of LO criticality tasks, or prolonging the execution of HI criticality tasks. This is the most simplistic scenario and relates directly to the analysis, however, if the level of pessimism is still high for the $C(LO)$ execution times this approach may lead to a significant amount of CPU idle time. In addition, LO criticality tasks are suspended for any deviation over $S^{\max}(j)$, no matter how small. Such a deviation might be tolerable, but the static nature of the implementation assumes LO criticality tasks are dropped. An approach, which could be implemented to maintain the guarantees of the offline analysis but provide a more flexible run-time implementation, is the barrier protocol

[2]Any execution beyond C(HI) values is considered erroneous behaviour.

[41], as introduced in Section 2.6. A barrier could be used in this scenario by waiting for all cores to signal completion of execution for the HI criticality mode, once done the barrier may release the LO criticality work for execution. While the offline analysis is still conducted to assure overall schedulability both in the standard and HI criticality modes, the barrier allows an element of dynamicism. HI criticality work can finish execution at any time, either before or after $S^{\max}(j)$, and the barrier will still release LO work. Clearly LO criticality work is not guaranteed to complete if the barrier invocation is after $S^{\max}(j)$. However, as $C(HI)$ predictions are often overly pessimistic, there may well be cases where $S^{\max}(j)$ is only exceeded by a small amount giving plenty of remaining execution time for LO criticality work to also complete. In the remainder of this work, we discuss the use of the barrier, however, both the static and barrier approaches are applicable.

We begin by investigating the allocation problem using heuristics on a simplified allocation problem. We assume that the minor cycle is equal to the major cycle ($T^F = T^M$), thus removing a dimension of allocation. The heuristics are compared to an optimal ILP based allocation technique (optimal in the sense that if a permissible allocation exists, it will be found). The later half of this chapter removes the minor cycle restriction and considers the allocation of a full mixed criticality cyclic executive with multiple cores and minor cycles.

3.1 Single Minor Cycle

As the allocation problem presented by a mixed criticality cyclic executive with many minor cycles and criticality levels is complex, we simplify our initial investigation to consider only a single minor cycle. In this case the minor cycle is equal to the major cycle $T^F = T^M$, however, we still consider a multi-core platform, thus allocation is across cores only. We illustrate this restriction in Figure 3.1 which illustrates single minor cycle but multiple cores (shown as C_1 and C_2).

With this simplification comes a number of additional assumptions:

- We assume that all task periods are equal to the length of the minor cycle, $T_i = T^F$.

Figure 3.1: A CE where $T^F = T^M$.

- We assume that all work of any task must complete within the minor cycle T^F. As $T^F = T^M$ each cycle is independent of each other and provides a fresh start.

We begin investigating task allocation to the single cycle model by considering some heuristic allocation techniques.

3.1.1 Task Allocation: Heuristics

Given the task allocation problem is known to be NP-hard for even a single processor [8], heuristic allocation techniques are often considered. This work considers a number of heuristic allocation techniques, the aim was to investigate the impact of using the barrier protocol on a multi-core, single minor cycle, cyclic executive. We considered three allocation techniques:

1. First Fit (FF): A common heuristic which attempts to allocate work onto the first core with available capacity. Starting at the highest criticality level, each task is allocated to the first core with enough spare capacity. The cores are numbered $1...x$, this ordering is used by the heuristic.

2. Worst Fit (WF): Another well-known heuristic, worst fit attempts to allocate work to the core with the largest spare capacity. Starting again at the highest criticality level, each task is allocated to the core with greatest spare capacity.

3. First Fit with Branch & Bound (FFBB): A combination of the common First Fit heuristic and a Branch & Bound search. The aim of this technique is to reduce the point S^{max} such that cores are idle for as little time as possible while waiting to execute the next criticality level. An initial allocation of First Fit is performed and the largest and smallest points of $S(i)$ (where i is the core)

are found. These points of $S(i)$ are used to create a goal S^{max}, the First Fit allocation is re-run with this goal as its limitation. If successful the point of S^{max} is reduced and if unsuccessful it is increased. A branch & bound search is performed to locate the minimum S^{max} value, thus discovering the earliest possible criticality switching point.

We undertook an experimental investigation to understand the effectiveness of each heuristic.

3.1.2 Experiment: Investigating the Impact of the synchronised criticality switching (Barrier Protocol)

These initial experiments are designed to assess the effectiveness of our allocation heuristics and to consider the impact of execution strictly separated by criticality level. We generate sets of random tasks with a variety of parameters to explore the impact. The general setup of all experiments is described below:

Setup

The experiments were undertaken on a large number of synthetically generated task sets. Per 5% utilisation increase, 1000 different task sets were generated. Matlab [62] was used to generate the tasks and to implement the heuristic allocation techniques, details of the setup are as follows:

- Task utilisations were generated using UUniFast, an algorithm presented in [25] which provides an unbiased distribution of utilisation values, following standard practice in synthetic task set generation.
- Task periods were set equal to the minor cycle length, in this case 25, $T = T^F = 25$. *(In reality 25 might equal 2500 in the implementation, typically this reduces the chance of a low utilisation value producing a very small execution time)*
- Deadlines were set equal to periods. $D_i = T_i$.
- The LO execution times of each task were produced as follows: $C_i(LO) = U_i/T_i$

- For tasks with a criticality greater than the lowest, their HI execution times were determined by $C_i(L_i) = C_i(LO) * CF$ - CF is the criticality factor, a random value between 1.2 and 2.

In these experiments (and others in this thesis) we make use of a metric called Weighted Schedulability [22]. This metric provides a single comparative value of schedulability allowing for the variation of more factors to investigate scalability, in essence we may present a three dimensional graph in two dimensions making the data easier to digest. An entire standard schedulability plot with increasing utilisation is summarised as a single weighted schedulability value allowing multiple experiments to be run while changing parameters such as the number of tasks or cores. We utilise the notation presented by [15] to describe Weighted Schedulability.

Weighted Schedulability[3]**:** $W_y(p)$ is the weighted schedulability of schedulability test y as a function of parameter p. For each p, we combine the results of all the tasksets τ generated for all of a set of equally spaced utilisation intervals. $S_y(\tau, p)$ is the binary result of schedulability test y for taskset τ with parameter p.

$$W_y(p) = (\sum_{\forall \tau} u(\tau) \cdot S_y(\tau, p) / \sum_{\forall \tau} u(\tau) \tag{3.1}$$

Results

A number of different experiments were undertaken, these are described in the following:

Experiment One

Parameters:

- 20 tasks were generated per task-set.
- Allocation was made to a 2 core platform.
- Tasks were evenly distributed over two criticality levels.

[3] As described by [15].

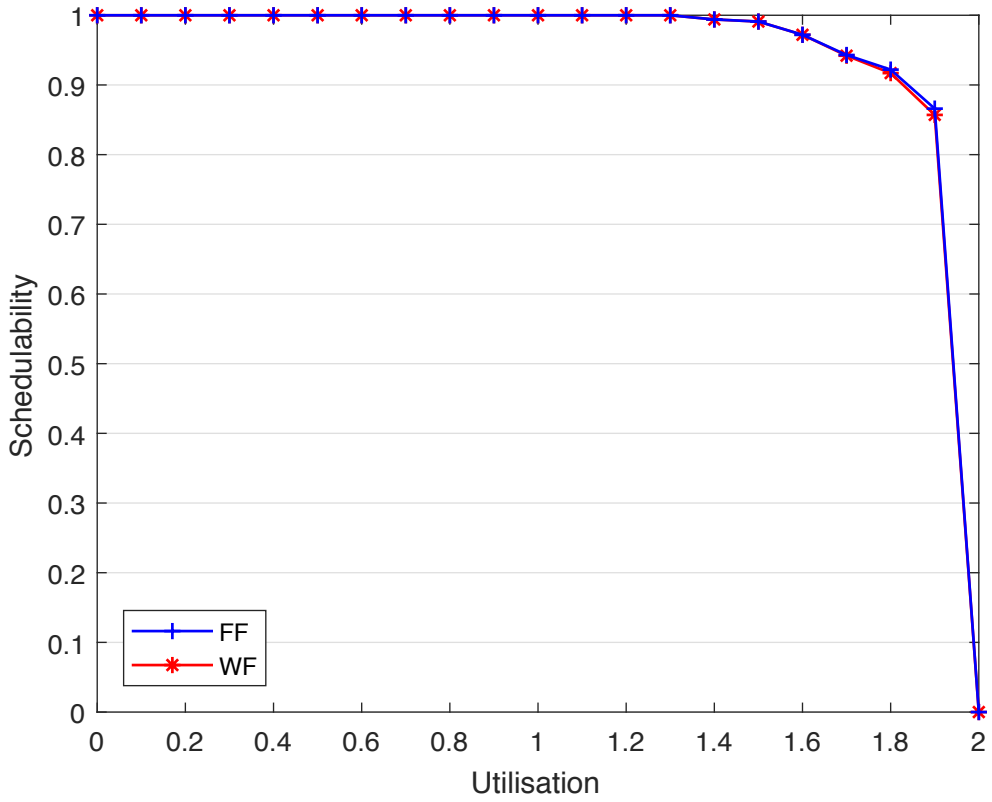


Figure 3.2: An illustration of the performance of FF and WF with no barrier protocol.

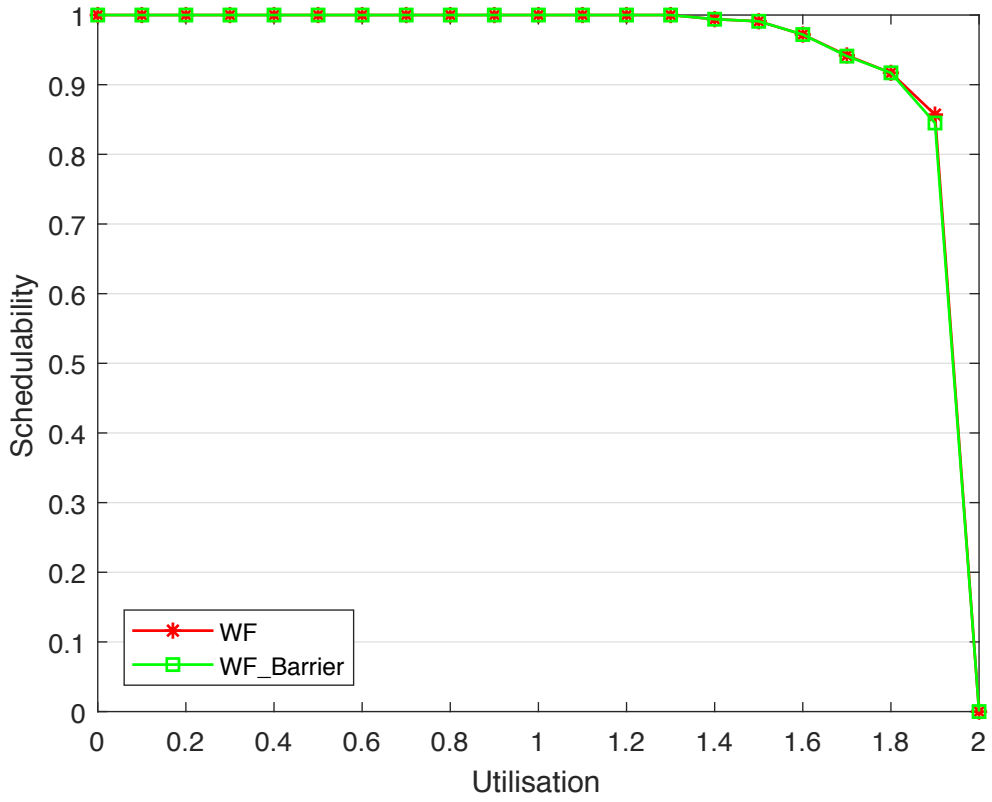


Figure 3.3: The impact of the barrier protocol on WF.

This initial experiment is designed to investigate the impact of using the barrier protocol. We begin by comparing FF and WF with no barrier protocol active (FFBB can't be included in this plot as the barrier is intrinsic to its function). This comparison is illustrated in Figure 3.2.We illustrate that both FF and WF are able to schedule a high percentage of the total task sets FF out-performs WF slightly. Not un-expectedly we observe good allocation performance with no barrier requirements. We now compare WF with and without the barrier in Figure 3.3.

We observe very little impact to schedulability as a result of the barrier. This is likely due to the even distribution of task sets produced by WF, as such, when all HI work is allocated, the difference in barrier invocation points across all cores is likely to be very small. Therefore only a very small impact is observed with the introduction of the barrier.

The same comparison between barrier and no barrier is performed for FF, the plot is presented in Figure 3.4.

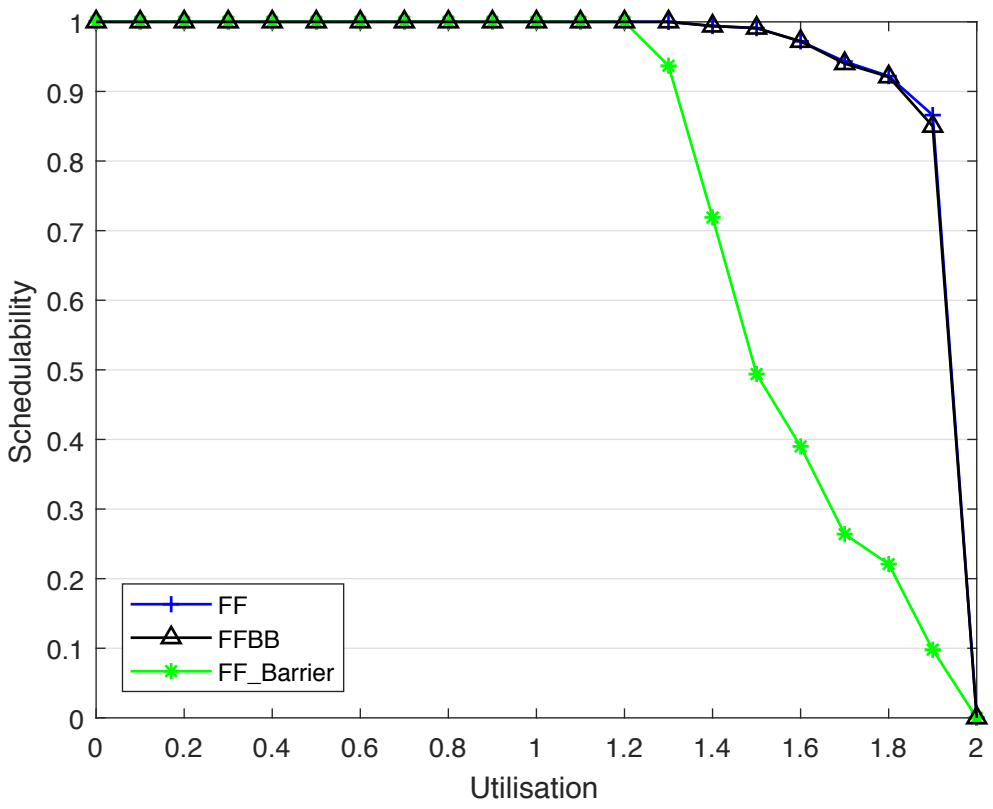


Figure 3.4: The impact of the barrier protocol on FF.

It is immediately clear that FF is heavily impacted by the introduction of a barrier. This is due to the way in which FF will allocate all HI work to a single core, pushing

the invocation of the barrier later causing a significant amount of lost utilisation on other cores. A solution to FFs performance is to introduce FFBB which performs an initial pass of FF then uses a branch & bound search to minimise the barrier invocation. We observe that FFBB performs very well and remains close to the performance of FF without the barrier implementation.

Finally, we plot WF and FFBB, the best performing approaches when the barrier is introduced. This is illustrated in Figure 3.5.

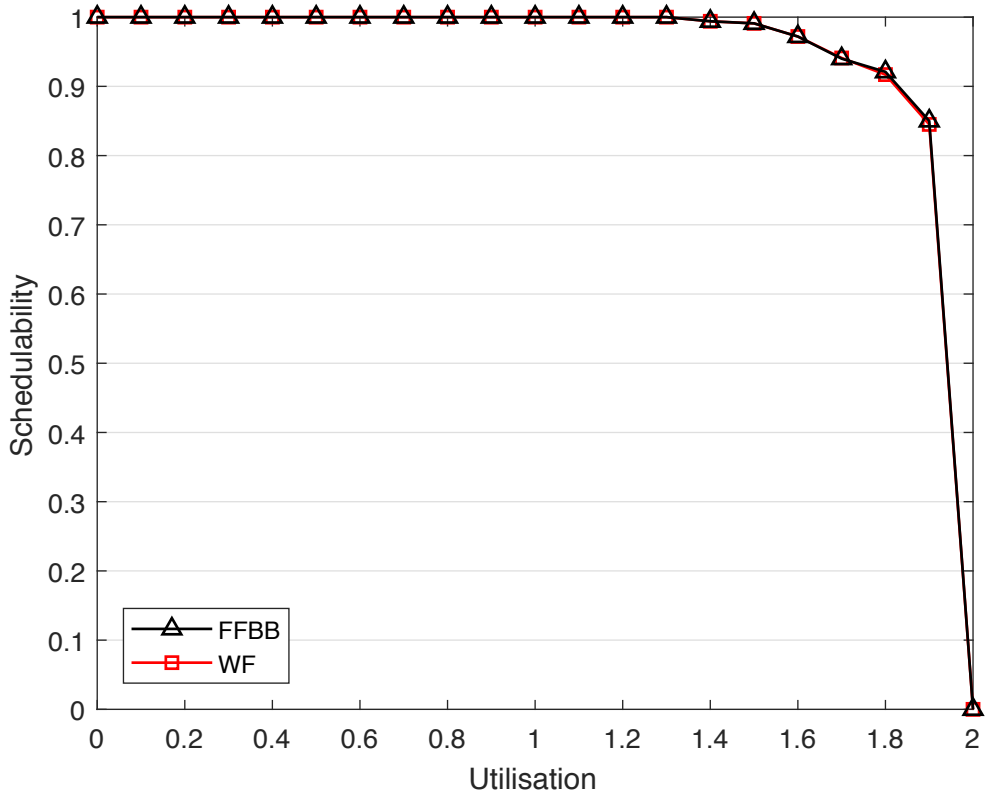


Figure 3.5: WF compared with FFBB.

We see very good performance from both approaches with FFBB taking a slight edge in schedulability at the higher utilisations.

Experiment Two

Parameters:

- Multiple experimental runs were performed, each increasing the number of tasks per set in increments of 20, from 20 to 100.
- Allocation was made to a 2 core platform.
- Tasks were evenly distributed over two criticality levels.

This experiment scales the number of tasks and investigates any change in the

allocation performance. The results of this plot can be seen in Figure 3.6. This plot uses Weighted Schedulability to present the change in relative overall schedulability as the number of tasks is increased. We compare WF with FFBB as these heuristics perform the best with the barrier implemented. The results are shown in Figure 3.6.We see little change in schedulability, with both approaches performing well.

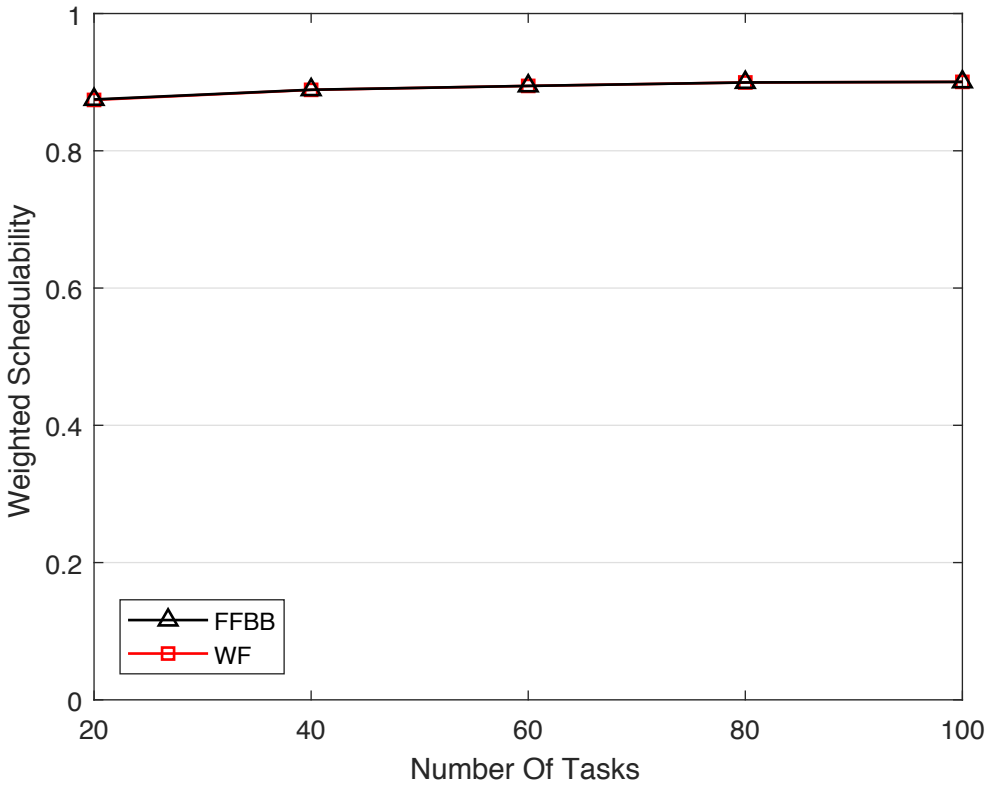


Figure 3.6: A plot illustrating the performance of WF and FFBB as the number of tasks is increased.

Experiment Three

Parameters:

- Multiple experimental runs were performed, each increasing the number of cores per set in increments of 2, from 2 to 8.
- 20 tasks were generated per task-set.
- Tasks were evenly distributed over two criticality levels.

Experiment three investigates scalability by cores. We present the results in Figure 3.7. Again we see both FFBB and WF with similar success. This plot illustrates a rapid decline in schedulability as the number of cores available increases. This is down to each task set only containing 20 tasks, with an increasing core count,

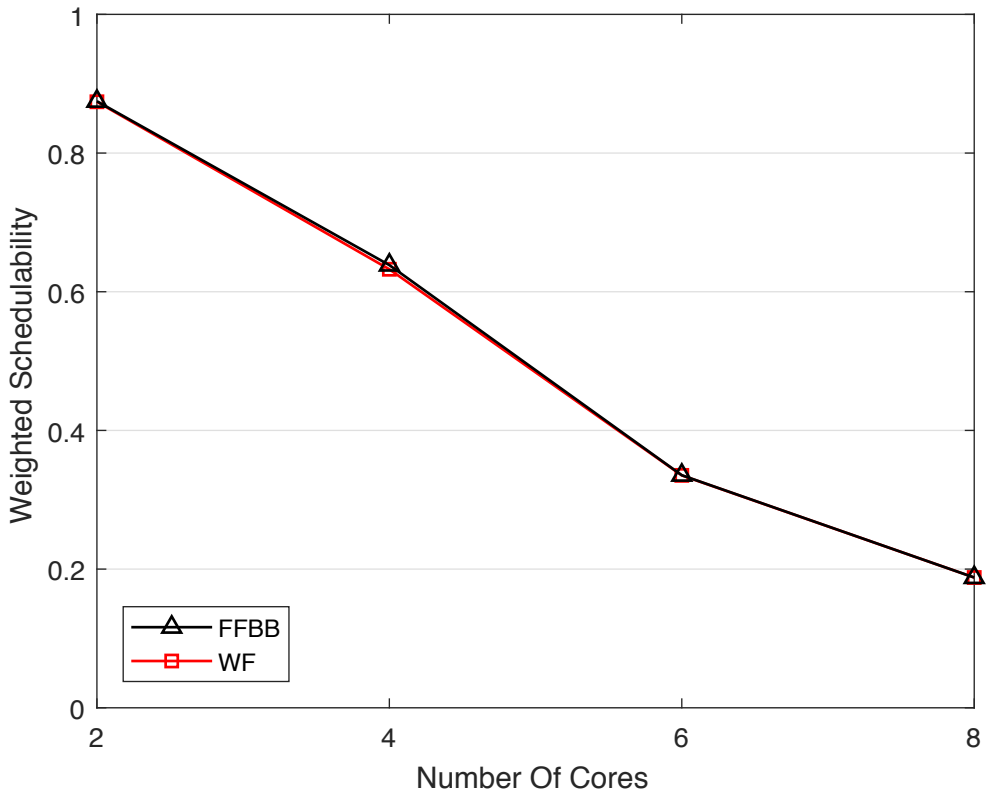


Figure 3.7: A plot illustrating the performance of each technique as the number of cores is increased.

these task sets must be generated at even greater utilisations. As the number of tasks is not high, some of these generated tasks may be given very large WCETs, to the extent that task sets become unschedulable rather quickly when compared to the 2 core plot. It should be noted that weighted schedulability gives a comparable measure of performance, a 4 core platform is still able to schedule more work than a 2 core, but uses proportionally less of its available platform.

To investigate this further we considered a number of scenarios with different numbers of cores and tasks. These were as follows:

- 20 tasks, 2 cores
- 20 tasks, 8 cores
- 50 tasks 8 cores
- 100 tasks, 8 cores

All experiments contained two levels of criticality. As these values do not follow on logically from each other, they are not plotted in the typical graph. Again the results are presented as weighted schedulability values and are presented in Table 3.1

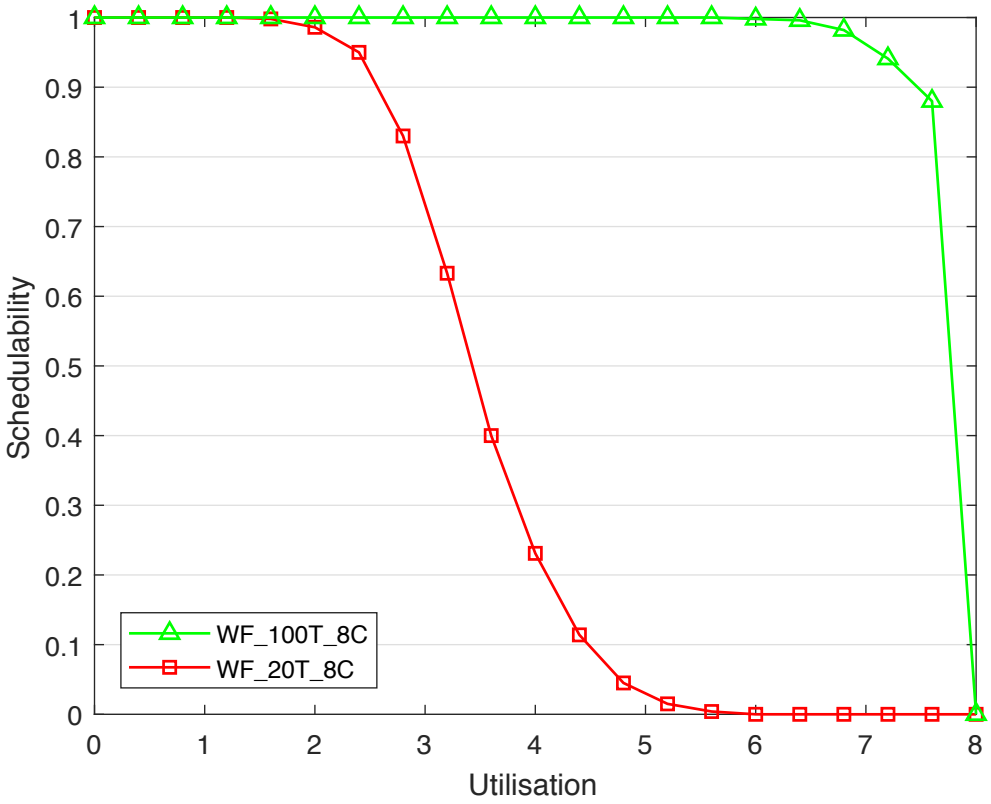


Figure 3.8: A plot illustrating the platform utilisation improvement when the number of tasks is increased alongside the number of cores.

Experiment	WF	FFBB
20 Tasks 2 Cores	0.87	0.87
20 Tasks 8 Cores	0.19	0.19
50 Tasks 8 Cores	0.63	0.64
100 Tasks 8 Cores	0.89	0.89

Table 3.1: Weighted Schedulability Results.

It is clear that a higher core count platform benefits greatly from greater granularity in the number of tasks to make best use of its increased processing potential. The weighted schedulability value for 8 cores and 20 tasks is 0.19 for FFBB, this increases dramatically to 0.77 when the number of tasks is increased to 100. This case is illustrated as a standard *utilisation X schedulability* plot in Figure 3.8.

Summary

These experiments have illustrated that the impact of introducing a barrier synchronisation mechanism is relatively low. We examined the effectiveness of WF, FF and FFBB:

- Worst fit suffers little impact with the introduction of synchronised criticality switching, due to its intrinsic desire to evenly distribute tasks across cores.

- While first fit is greatly impacted by synchronised criticality switching, the loss in schedulability can be mitigated through the implementation of FFBB.

- FFBB did not prove any more effective than WF. Given that WF a common and well known heuristic we chose to use it in later experiments in this Chapter.

Overall while we observe some impact involved with the synchronised criticality switching of criticality levels, the loss in schedulability is relatively minor. From this minor loss in schedulability we gain the separation, robustness and dynamicism which are fundamental to our approach.

3.1.3 Task Allocation: Integer Linear Programming

The use of Linear Programming in this early stage of work was driven by the question: *How well do our heuristic allocation methods perform in comparison to an optimal allocation technique?* Integer Linear Programming provides such an optimal solution, such that if a feasible allocation of tasks to cores exists, the solver will find it. In addition we sought to understand whether it was feasible to utilise a Linear Programming solver efficiently for this allocation problem. The Linear Programming solver Gurobi [45] was used throughout this work. The structure of our initial ILP models will be defined using Gurobi's '.lp' format, a more human readable format in comparison to the widely used '.mps'. Each section of the model will be described along with example constraints to illustrate functionality.

Objective Function (Maximise/Minimise)

The objective function describes the aim of any optimisation to be performed by the model. This takes the form of a statement which must be minimised or maximised. However, in this case the objective function is left empty, as such our models are feasibility tests rather than optimisations. We are only concerned with discovering

whether or not a feasible allocation of tasks exists, rather than attempting to optimise a particular feature. To be consistent in illustrating the structure of the model we include a block as part of our abstract model representation, see Figure 3.9:

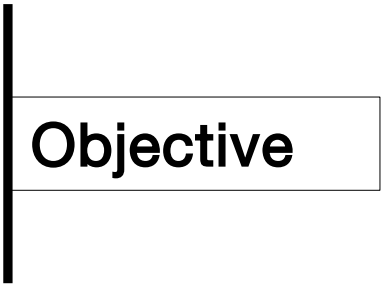


Figure 3.9: Abstract Diagram: Objective.

Constraints (Subject To)

Constraints make up the fundamental description of the problem which the Linear Programming solver uses to produce a solution. These take the form of linear inequalities. In our models the constraints section defines how all tasks in a given system may be allocated. To illustrate this section we must describe how the possible allocations of tasks are represented in our models.

We define a binary variable for each task at each possible location to which it may be allocated. This provides a basis for the construction of constraints to restrict the number of variables set to 1. When a location variable is set to 1 in our models, this indicates that the task is allocated to execute at that particular location in the system, if set to 0 the task is not allocated in that location. Our variables may be described as follows:

$$Q[taskNumber]\_[core]$$

For an example task, τ_1, the variables required to represent its possible schedulable locations of a 4 core platform are:

Q1_1 , Q1_2 , Q1_3 , Q1_4

One Q variable is included for τ_1 for each core in the system. These same variables are shown in Figure 3.10 to better illustrate which location each represents.

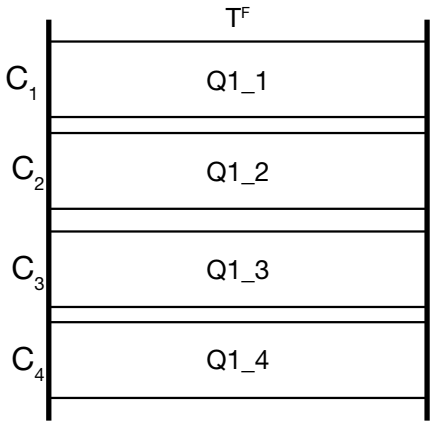


Figure 3.10: Location variables in the locations they represent.

Clearly the task may only be scheduled on one of the four available cores, this leads into the first set of constraints. Such constraints are concerned with ensuring that a task may only be scheduled once on one of the available cores. Consider the following constraint for our example τ_1:

Q1_1 + Q1_2 + Q1_3 + Q1_4 = 1

This constraint ensures that the sum of those binary variables must be equal to one, thus the task must be scheduled only on a single core as just one variable may be set equal to 1. We include such constraints as a sub-block of the constraints named 'TaskLocation', this is shown in Figure 3.11:

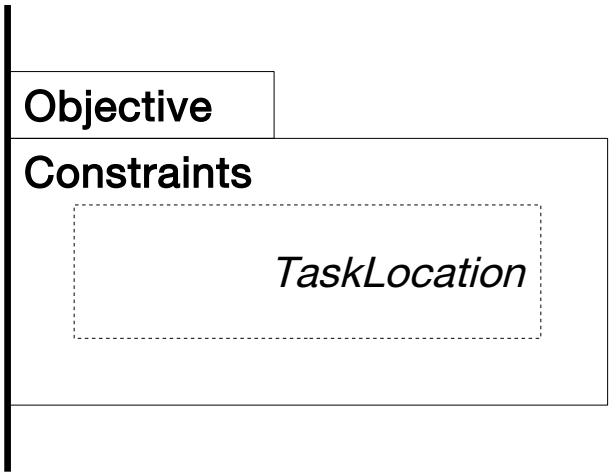


Figure 3.11: Abstract Diagram: Objective, Constraints.

The second fundamental set of constraints is implemented to ensure that if a task is allocated on a particular core, it is schedulable. There are three parts to these constraints: Firstly HI criticality tasks have each of their binary variables

multiplied by their $C(HI)$ WCETs, if a binary is set to 1 the WCET is included in the calculation, if 0 it is not. In this way one constraint is defined for each CPU core. Given 2 HI criticality tasks τ_1 & τ_2 and the same 4 core platform, the constraints look as follows:

$$C_1(HI) \times Q1\_1 + C_2(HI) \times Q2\_1 \leq T^F$$
$$C_1(HI) \times Q1\_2 + C_2(HI) \times Q2\_2 \leq T^F$$
$$C_1(HI) \times Q1\_3 + C_2(HI) \times Q2\_3 \leq T^F$$
$$C_1(HI) \times Q1\_4 + C_2(HI) \times Q2\_4 \leq T^F$$

These constraints ensure that in the worst case HI criticality tasks are able to execute to their HI WCET values on the core tha they are scheduled. The second set of constraints looks similar but uses the LO WCET values for the HI criticality tasks in order to calculate the synchronised switching point.

$$C_1(LO) \times Q1\_1 + C_2(LO) \times Q2\_1 + X \leq T^F$$
$$C_1(LO) \times Q1\_2 + C_2(LO) \times Q2\_2 + X \leq T^F$$
$$C_1(LO) \times Q1\_3 + C_2(LO) \times Q2\_3 + X \leq T^F$$
$$C_1(LO) \times Q1\_4 + C_2(LO) \times Q2\_4 + X \leq T^F$$

The variable X is used to represent the spare capacity of each core which LO criticality tasks may be scheduled in. Using the same X variable for each calculation provides an implementation of the barrier protocol ensuring that the spare capacity, and thus the switching point (S^{max}), is the same across all cores. The third set of WCET constraints checks the schedulability of all LO criticality tasks. The constraints for LO criticality tasks τ_3 & τ_4 would be as follows:

$$C_3(LO) \times Q3\_1 + C_4(LO) \times Q4\_1 - X \leq 0$$
$$C_3(LO) \times Q3\_2 + C_4(LO) \times Q4\_2 - X \leq 0$$
$$C_3(LO) \times Q3\_3 + C_4(LO) \times Q4\_3 - X \leq 0$$
$$C_3(LO) \times Q3\_4 + C_4(LO) \times Q4\_4 - X \leq 0$$

In this way LO tasks are required to execute in the time after the barrier protocol has triggered. This time is represented by variable X, thus if the sum of the LO execution is less than or equal to X it is schedulable. The three sections presented above are the *WCETConstraints* illustrated in Figure 3.12:

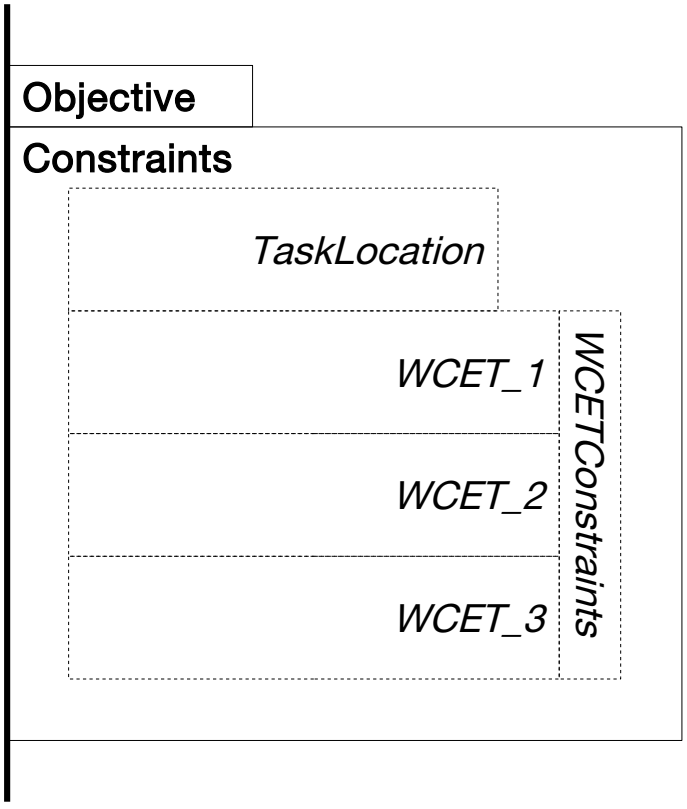


Figure 3.12: Abstract Diagram: Objective, all constraints.

Bounds

The bounds section for these models is simple, as all of the task location variables are binaries, the only integer variable is X. X is constrained as so:

$$X <= T^F$$

A lower bound of 0 is implicitly included.

Variables (Binaries/Integers)

The final section of a '.lp' model defines the variables based on their type, all task variables are defined as binaries and X is defined as an Integer. For example, τ_1 & τ_2 and X are defined as follows:

```
Binaries
        Q1_1  Q1_2  Q1_3  Q1_4  Q2_1  Q2_2  Q2_3  Q2_4
Integers
        X
End
```

Model Overview

With the constraints described, Figure 3.13 shows an overview of a typical ILP model describing a set of tasks seeking schedulability of a single cycle mixed criticality multi-core cyclic executive.

3.1.4 Experiment: Heuristics vs ILP

The purpose of this experiment was to investigate how well the heuristics performed in comparison to the *optimal* ILP. This time we compared all approaches utilising the synchronised switching mechanism (barrier protocol). While we recorded execution timing data, this is discussed in Section 3.2.2. We begin by describing the experimental setup.

Setup

Both the heuristic techniques and an ILP model generator were implemented in Matlab. ILP models may be generated and evaluated while within the Matlab environment using the Gurobi integration tools. We generated 1000 task sets per 5% increase in utilisation. The setup of this experimentation is detailed below:

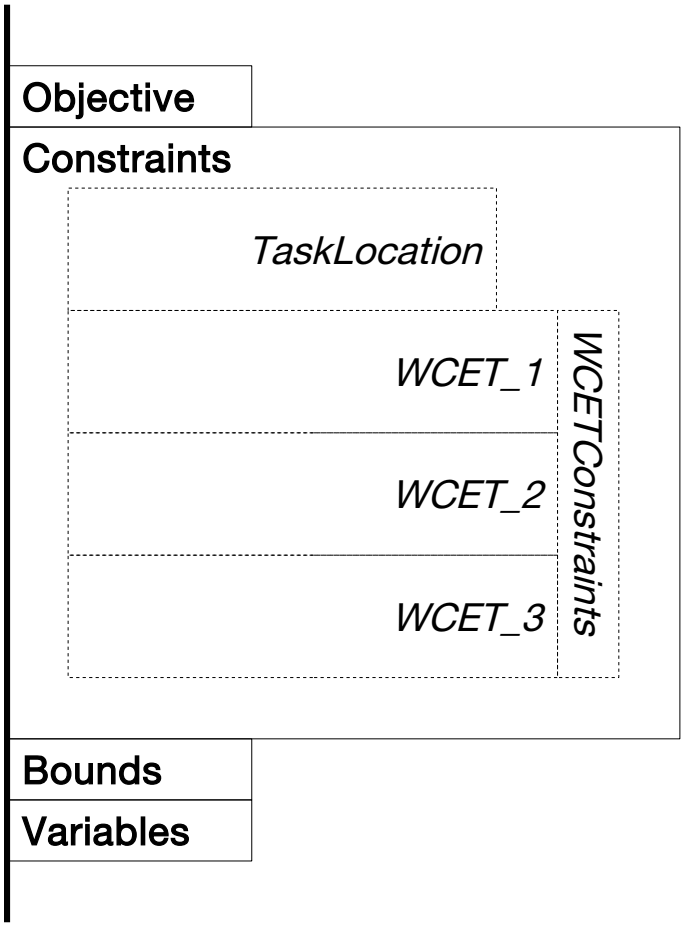


Figure 3.13: Abstract Diagram: Full (single-cycle).

- Task utilisations were generated using UUniFast, an algorithm presented in [25] which provides an unbiased distribution of utilisation values, following standard practice in synthetic task set generation.
- Task periods were set equal to the minor cycle length, in this case 25, $T = T^F = 25$. *(In reality 25 might equal 2500 in the implementation, typically this reduces the chance of a low utilisation value producing a very small execution time)*
- Deadlines were set equal to periods. $D_i = T_i$.
- The LO execution times of each task were produced as follows: $C_i(LO) = U_i/T_i$
- For tasks with a criticality greater than the lowest, their HI execution times were determined by $C_i(L_i) = C_i(LO) * CF$ - CF is the criticality factor, a random value between 1.2 and 2.

- Timing data was recorded to find the average time each heuristic took to find a solution. (detailed analysis of timing data conducted in Section 3.2.2).

- The barrier protocol was implemented for all allocation techniques.

Results

The experimental results consist of a standard schedulability plot and a number of plots investigating the scalability of each approach using the Weighted Schedulability metric.

Experiment One

Parameters:

- 20 tasks were generated per task-set.
- Allocation was made to a 2 core platform.
- Tasks were evenly distributed over two criticality levels.

The first experiment was performed to provide the standard plot of utilisation vs schedulability. The result of this experiment is shown in Figure 3.14. As expected FF is the worst performing herustic, as expected due to its tendency to fill up a single core with HI criticality work. Both WF and FFBB perform very close to the optimal ILP allocation providing a very good approximation.

Experiment Two

Parameters:

- Multiple experimental runs were performed, each increasing the number of tasks per set in increments of 20, from 20 to 100.
- Allocation was made to a 2 core platform.
- Tasks were evenly distributed over two criticality levels.

The second experiment explores the impact that increasing the number of tasks per set has on schedulability. Weighted schedulability is used to gauge the impact by summarising each individual experiment undertaken with varying task set sizes. The results of this experiment are shown in Figure 3.15.

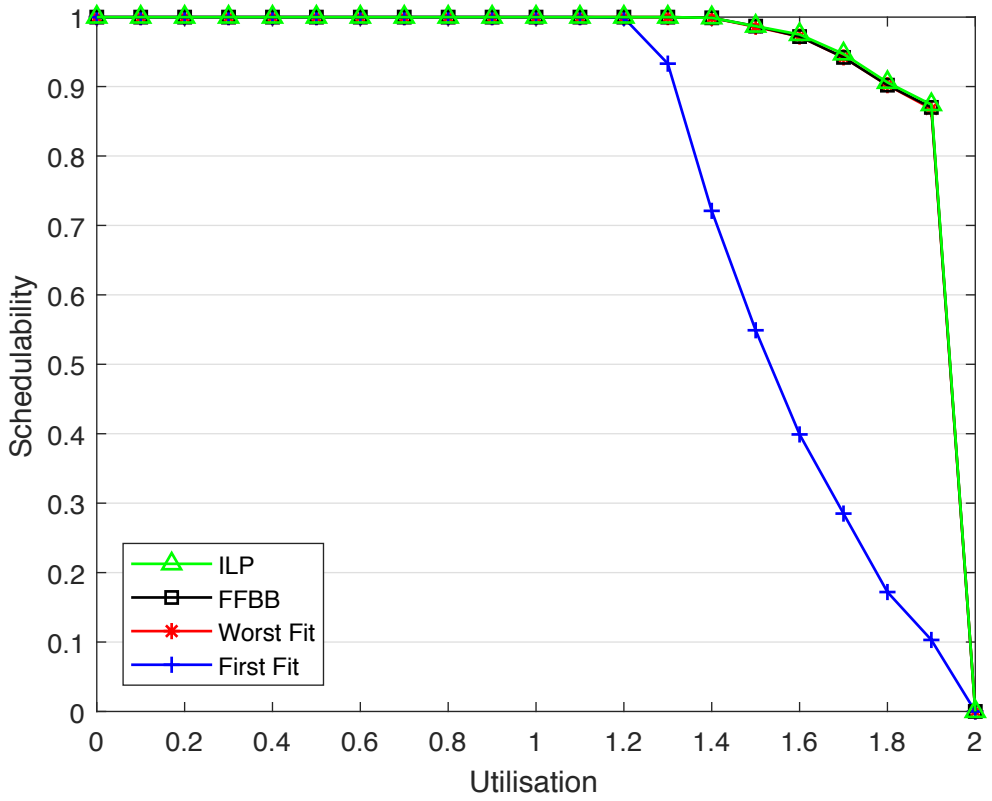


Figure 3.14: A graph illustrating the performance of the heuristic approaches compared to ILP where $T^F = T^M$.

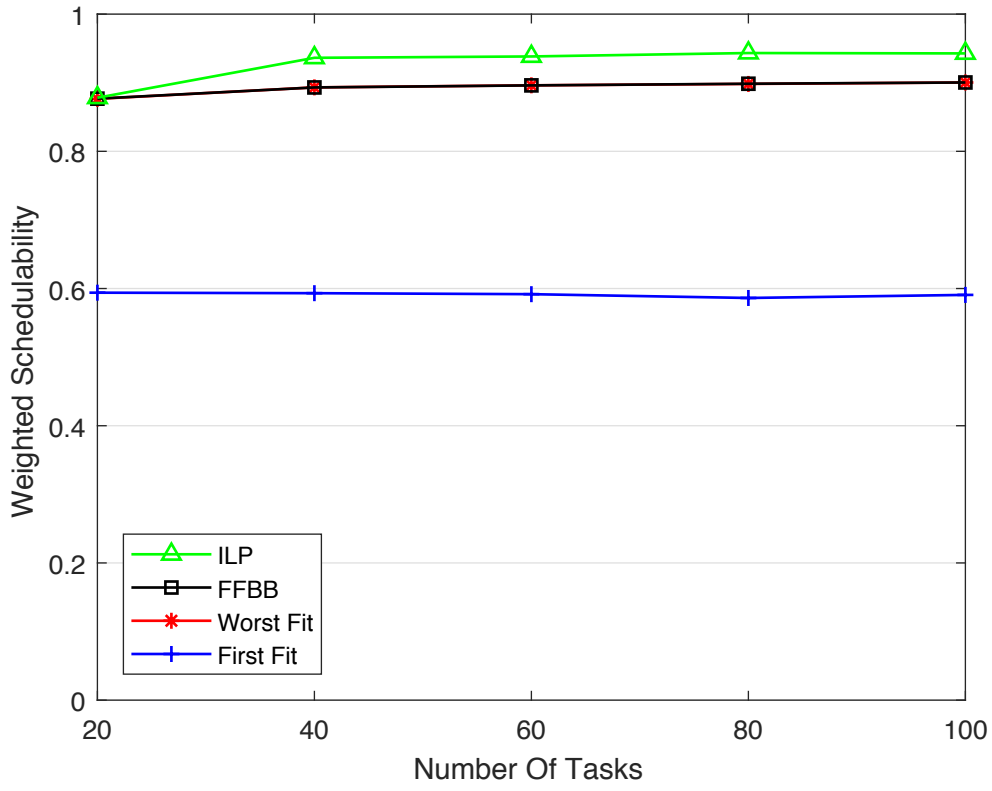


Figure 3.15: A graph illustrating the effect of increasing the number of tasks in each set.

This plot illustrates strong performance from FFBB and WF, however they do lose a small amount of schedulability when compared to the ILP approach. Given the greater granularity of tasks to allocate to 2 cores, ILP is able to schedule task sets even when the utilisation is equal to 2, while the heuristics cannot. The slight gain in schedulability for the ILP approach is entirely down to being able to schedule these task sets with very high overall utilisation.

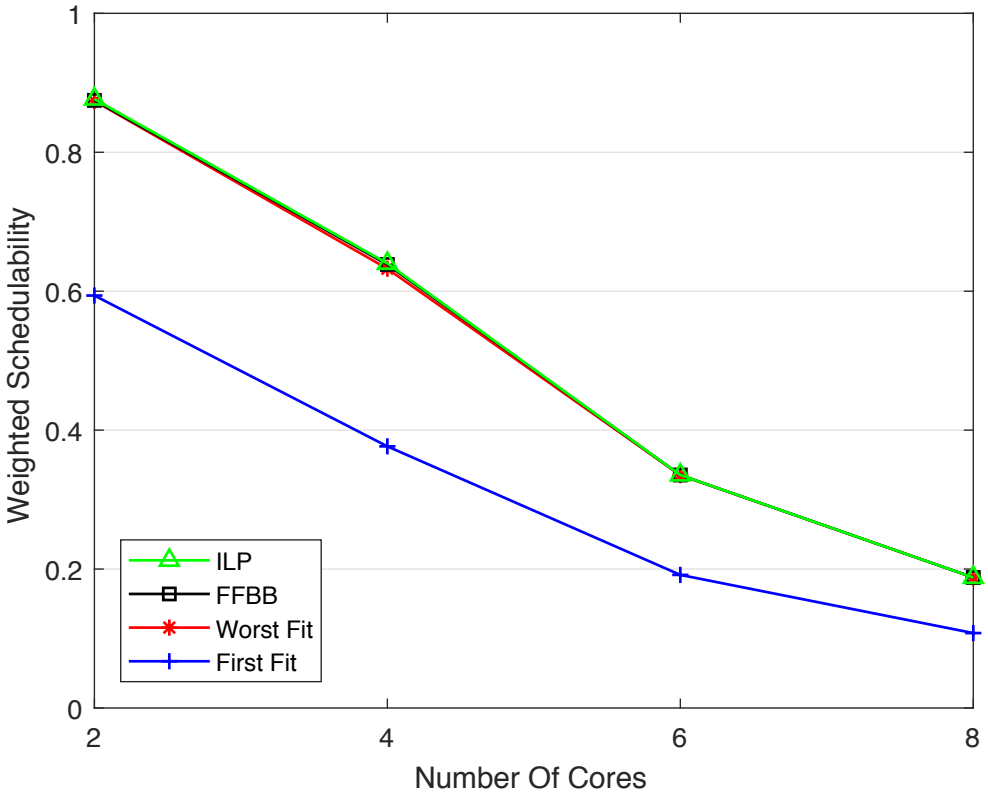


Figure 3.16: A graph illustrating the effect of increasing the number of cores to allocate to.

Experiment Three

Parameters:

- Multiple experimental runs were performed, each increasing the number of cores per set in increments of 2, from 2 to 8.
- 20 tasks were generated per task-set.
- Tasks were evenly distributed over two criticality levels.

Experiment three investigates the effect of increasing the number of cores on schedulability. The task set size is fixed at 20 and the number of criticality levels at 2. The results of this experiment use weighted schedulability and are presented

in Figure 3.16. As we saw previously (see Figure 3.7) increasing the number of cores without increasing the number of tasks to allocate has a negative impact on weighted schedulability. FF is clearly the worst performer, FFBB and WF are close to each other while ILP outperforms all approaches by a small margin. Otherwise the reduction in weighted schedulability is similar for each approach. Increasing the number of tasks would greatly improve this performance (as shown before in Figure 3.8).

Summary

In summary, the heuristics WF and FFBB are good approximations of the ILP solution while FF performs poorly. The poor performance of FF is intrinsic to its allocation policy, while WF and FFBB are reasonably close to the schedulabiliity of the optimal ILP approach. Given this result we move on to extend the model to include multiple minor cycles and investigate the performance.

3.2 Multiple Minor Cycles

A natural extension from the single cycle model is to introduce multiple minor cycles. This section builds upon the lessons learned when allocating single cycle cyclic executive and applies them to the multi cycle case. We extend the Integer Linear Programming model to support multiple minor cycles and compare its performance with the worst fit heuristic.

3.2.1 The Extended Model

The multi-cycle model assumes more than one minor cycle per major cycle. We assume that all minor cycles are of equal length, Figure 3.17 presents an overview:

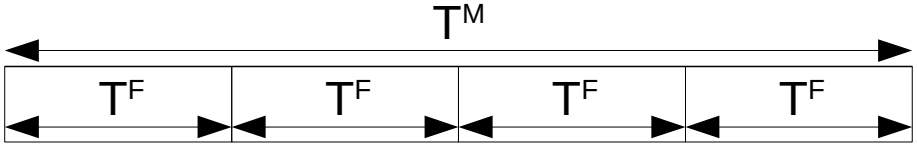


Figure 3.17: A cyclic executive with 4 minor cycles per major cycle.

τ	$C(LO)$	$C(HI)$	T	L_i
τ_1	5	10	25	HI
τ_2	5	10	25	HI
τ_3	10	15	50	HI
τ_4	15	20	100	HI
τ_5	5	-	25	LO
τ_6	5	-	25	LO
τ_7	10	-	50	LO
τ_8	10	-	100	LO

Table 3.2: Task Allocation: Example.

It is clear from Figure 3.17, that multiple T^F are present within T^M. While conceptually this extension is simple, some considerations must be made.

Tasks where $T > T^F$

Given multiple minor cycles, tasks are now permitted to have periods greater than the major cycle, however, they must follow the standard cyclic executive constraints and be a multiple of the minor cycle. In addition, each task, regardless of its period (whether it is equal to T^F or T^M), must complete its execution within a single minor cycle. Consider the task set in Table 3.2

These tasks have periods ranging from the minor cycle length $T^F = 25$ to the major cycle $T^M = 100$. The execution times for all tasks are $\leq T^F(25)$ and thus may complete within a single minor cycle. Figure 3.18 shows an example allocation of the task set from Table 3.2

This schedule illustrates how tasks of differing periods may be scheduled, some executing once every minor cycle, while some execute as little as once every major cycle. The periods utilised in this example are specifically chosen to allow for a cyclic executive schedule with a major cycle of 100 and a minor cycle of 25. This mimics common development practice, where such complimentary periods are enforced to ease the task allocation problem.

Extensions to the ILP Model

In order to support multiple minor cycles, a number of changes must be made to the constraints of an ILP model. We build upon the single cycle model described

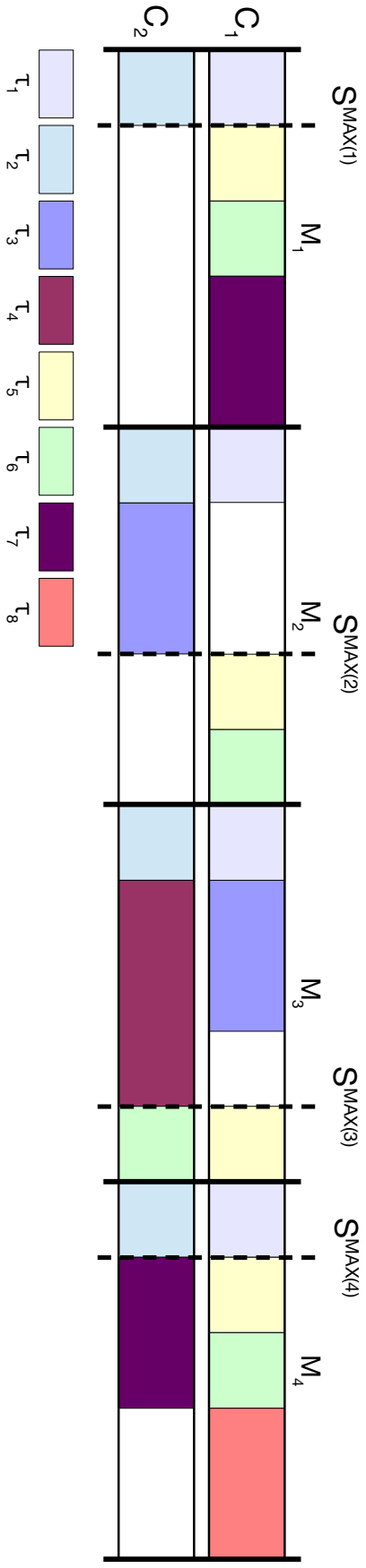


Figure 3.18: An example schedule (of Table 3.2) on a 2 core (C_1, C_2), 4 minor cycle ($M_1 ... M_4$) platform.

earlier in the chapter and explain these extensions.

Notation Change

To begin we must clarify a change in the notation to account for multiple minor cycles. As the binary variables representing each possible task location must now account for multiple minor cycles, their structure is altered accordingly. Each possible location that a task may be scheduled in is now represented by:

$$Q\{taskNumber\}\_\{CPU\_core\}\{MinorCycle\}$$

Thus, for task τ_1 on core 1 minor cycle 1 the variable would read:

$$Q1\_11$$

As before, we may illustrate the locations each variable represents using an extended version of Figure 3.10, this is shown in Figure 3.19:

	T^F	T^F	T^F	T^F
C_1	Q1_11	Q1_12	Q1_13	Q1_14
C_2	Q1_21	Q1_22	Q1_23	Q1_24
C_3	Q1_31	Q1_32	Q1_33	Q1_34
C_4	Q1_41	Q1_42	Q1_43	Q1_44

Figure 3.19: Location variables in the locations they represent extended to multiple minor cycles.

Task Location/Periodicity constraints

The first set of constraints presented, termed Task Location constraints earlier in the chapter, are now called periodicity constraints, as they seek to ensure each task executes at the correct frequency as well as indicating which core. To illustrate these possible constraints we will show how a generic task τ_i may be constrained in a system with 4 minor cycles per T^M.

- Where $T_i = T^F$: In this case one instance of τ_i must execute during every

minor cycle on one of the available cores. On a platform with 2 cores these statements would read:

$$Q_i\_11 + Q_i\_21 = 1$$
$$Q_i\_12 + Q_i\_22 = 1$$
$$Q_i\_13 + Q_i\_23 = 1$$
$$Q_i\_14 + Q_i\_24 = 1$$

- Where $T_i = T^M/2$: Here the task must execute once in the first two minor cycles and once in the second.

$$Q_i\_11 + Q_i\_21 + Q_i\_12 + Q_i\_22 = 1$$
$$Q_i\_13 + Q_i\_23 + Q_i\_14 + Q_i\_24 = 1$$

- Where $T_i = T^M$: Finally this task must only execute once every T^M:

$$Q_i\_11 + Q_i\_21 + Q_i\_12 + Q_i\_22 + Q_i\_13 + Q_i\_23 + Q_i\_14 + Q_i\_24 = 1$$

WCETConstraints:

Fundamentally little changes in this section, it is simply extended to account for an increase in the number of locations. Up to this point it has been reasonable to present the raw constraints, however as the number of tasks, cores or minor cycles involved increases, this becomes unwieldy very quickly. As such we define m to represent the minor cycle and c to represent the core.

- WCET_1: These constraints are repeated for each possible combination of m and c (for all cores in each minor cycle).

$$\forall i \sum_{i \in hct} C_i(HI) \times Q_i\_cm \leq T^F$$

Where hct is the set of high criticality tasks. To clarify for core 1 minor cycle

1 ($c = 1, m = 1$) the constraints for two HI criticality tasks, $\tau_i \& \tau_l$, would read:

$$C_i(HI) \times Q_i\_11 + C_l(HI) \times Q_l\_11 \leq T^F$$

- WCET_2: In this step, multiple X variables are now used to model the action of the barrier protocol during each minor cycle. Again the constraints are repeated for each possible combination of m and c.

$$\forall i \sum_{i \in hct} C_i(LO) \times Q_i\_cm + X_m \leq T^F$$

Thus the raw constraints where $c = 1$ and $m = 1$ are:

$$C_i(LO) \times Q_i\_11 + C_l(LO) \times Q_l\_11 + X_1 \leq T^F$$

- WCET_3: Finally LO criticality tasks must fit after the HI criticality work within the space in the X variable.

$$\forall i \sum_{i \in lct} C_i(LO) \times Q_i\_cm - X_m \leq 0$$

Where lct is the set of LO criticality tasks. Again the constraints for two LO criticality tasks $\tau_z \& \tau_x$ where $c = 1$ and $m = 1$ are:

$$C_z(LO) \times Q_z\_11 + C_x(LO) \times Q_x\_11 - X_1 \leq 0$$

While the structure of the model has not fundamentally changed, the overview is repeated below with minor updates in Figure 3.20:

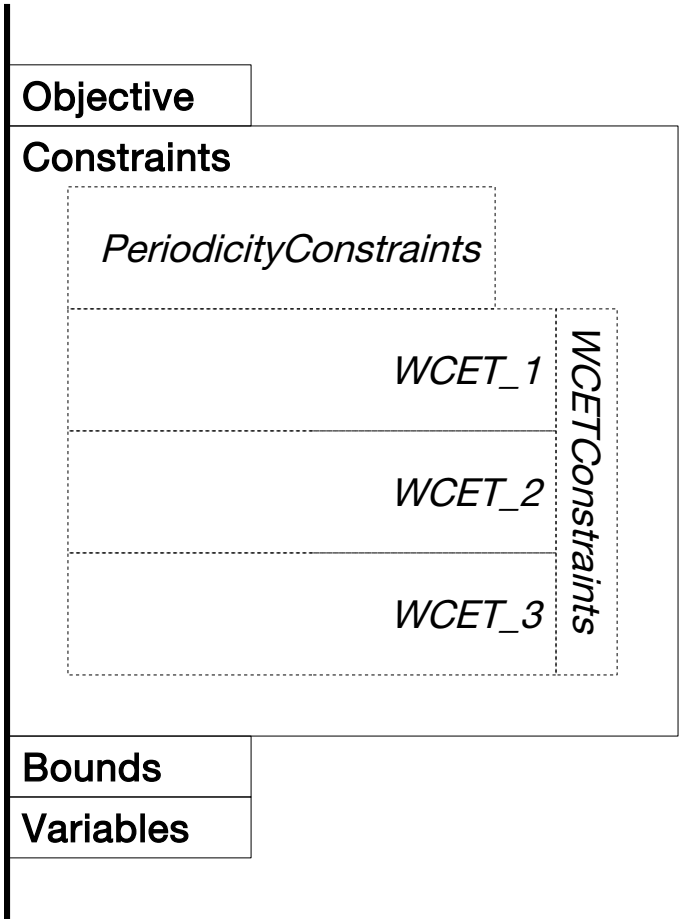


Figure 3.20: Abstract Diagram: Full (multi-cycle).

3.2.2 Experiment: Heuristics vs ILP - multi-cycle

With the model extended to multiple minor cycles some more experiments were performed to assess the performance of the Worst Fit when compared with the ILP model. The aim of this experiment is simply to assess the performance of all WF and ILP when allocating to multiple minor cycles. In addition, timing data is presented considering the time taken by each approach to find a solution.

Setup

A larger number of synthetically generated task sets were produced. 1000 task sets were generated per 5% utilisation increment. The standard features of the experimental setup are described below:

- Task utilisations were generated using UUniFast, an algorithm presented in

[25] which provides an unbiased distribution of utilisation values, following standard practice in synthetic task set generation.

- The minor cycle length was set at 25, with the major cycle length set at 100 ($T^F = 25, T^M = 100$)
- Task periods were selected at random from either 25, 50 or 100.
- Deadlines were set equal to periods. $D_i = T_i$.
- The LO execution times of each task were produced as follows: $C_i(LO) = U_i/T_i$
- For tasks with a criticality greater than the lowest, their HI execution times were determined by $C_i(L_i) = C_i(LO) * CF$ - CF is the criticality factor, a random value between 1.2 and 2.
- Timing data was recorded to find the average time each heuristic took to find a solution. All timing data was recorded on a 4 core Intel i7 4790K.
- The barrier protocol was implemented for all allocation techniques.

Additionally, we extended Worst Fit to be able to account for multiple minor cycles. Allocation begins at the highest criticality level, tasks are allocated to the core with the greatest available capacity, the range of cores available for this allocation is determined by the period of the task. If $T^M = 100$ and $T^F = 25$, a task with a period of 50 will be allocated to the core with the most spare capacity in the first two minor cycles and the same in the second two. Once the highest criticality level is allocated a value of S^{max} is found for each minor cycle, allocation of the next criticality level then begins and repeats the same process until all work is allocated.

Results

Experiment One

Parameters:

- 20 tasks were generated per task-set.
- Allocation was made to a 2 core platform.
- Tasks were evenly distributed over two criticality levels.
- 4 minor cycles were included per major cycle.

The first experiment is presented as a standard schedulability plot detailing the performance of both techniques. The results are shown in Figure 3.21:

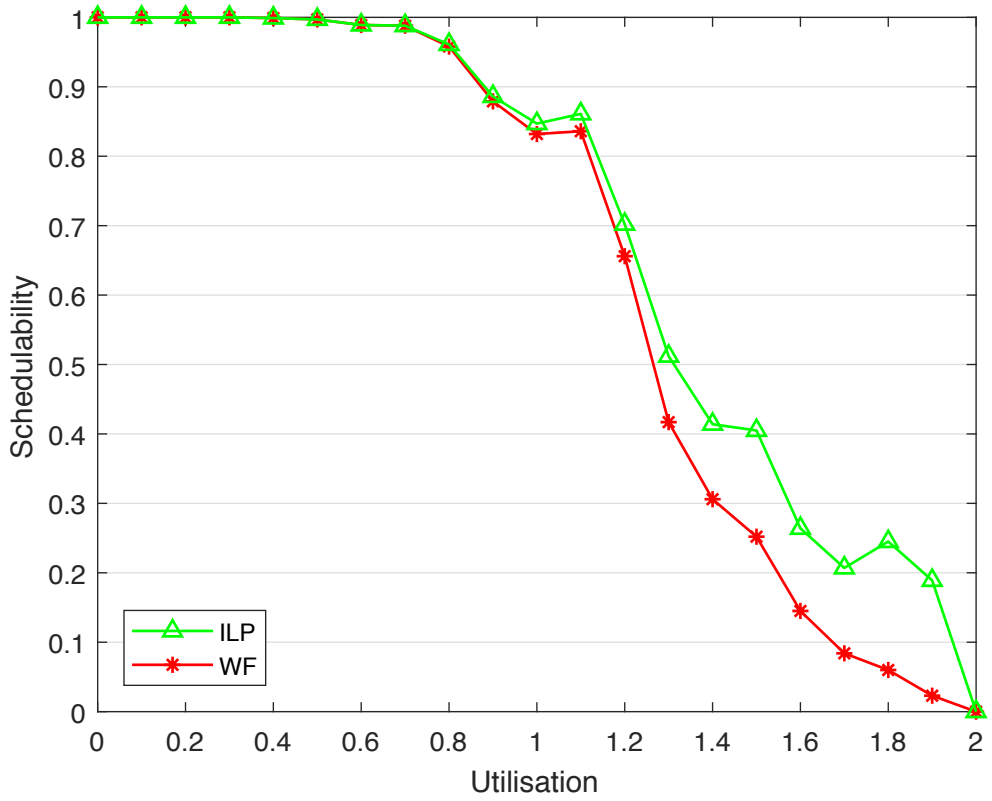


Figure 3.21: A comparison of the performance of all allocation approaches.

We observe WF performing significantly worse than in the single-cycle case. This result was expected as WF allocates tasks in order, an allocation for the previous criticality level may be undesirable for the next, the heuristic is not able to re-allocate. However the ILP approach does not rely on a sequential allocation of tasks, as such it is able to allocate tasks at all criticality levels in the most optimal fashion.

Experiment Two

Parameters:

- Multiple experimental runs were performed, each increasing the number of tasks per set in increments of 20, from 20 to 100.
- Allocation was made to a 2 core platform.
- Tasks were evenly distributed over two criticality levels.
- 4 minor cycles were included per major cycle.

This experiment investigates the effect of scaling the number of tasks from 20 per set to 100. We present the results in 3.22.

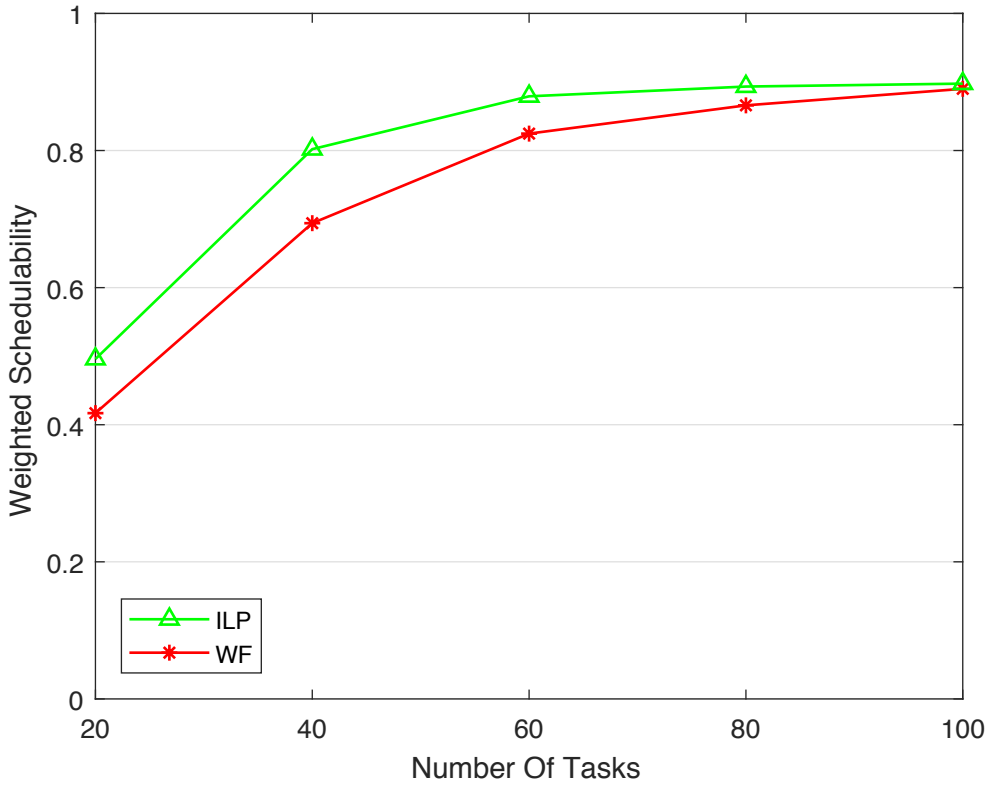


Figure 3.22: A plot illustrating the performance as the number of tasks per set is increased.

Figure 3.22 illustrates that scaling the number of tasks provides a significant boost in schedulability for both ILP and WF. While ILP always out performs WF, we see the results narrowing as the number of tasks is increased. The general increase is due to an increased granularity in the allocation, the utilisation of each task set at each set is the same regardless of the number of tasks, thus the execution time of a task in a task-set of size 100 is likely to be relatively small. This results in task sets which are easier to allocate, this ease of allocation explains the gains made by WF and the overall gains made by both approaches.

Experiment Three

Parameters:

- Multiple experimental runs were performed, each increasing the number of cores per set in increments of 2, from 2 to 8.
- All task sets contained 20 tasks.
- Tasks were evenly distributed over two criticality levels.
- 4 minor cycles were included per major cycle.

We then investigated the performance when the number of cores are scaled. The results are shown in Figure 3.23.

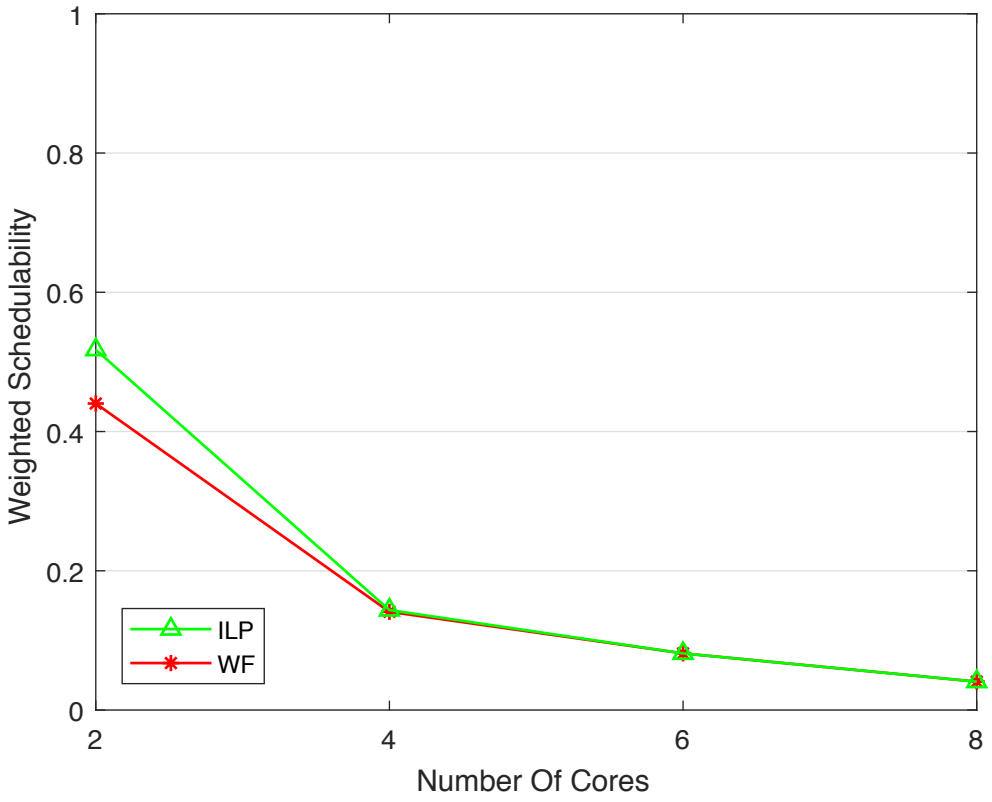


Figure 3.23: A plot illustrating the performance as the number of cores is increased.

As we have observed previously (see Section 3.1.2) if the number of cores is scaled up without an increase in the number of tasks a significant loss in schedulability is observed. We illustrate this with two plots in Figure 3.24. These results again illustrate how additional locations for allocation, created by increasing the core count, require a suitably large number of tasks in order to be efficiently allocated.

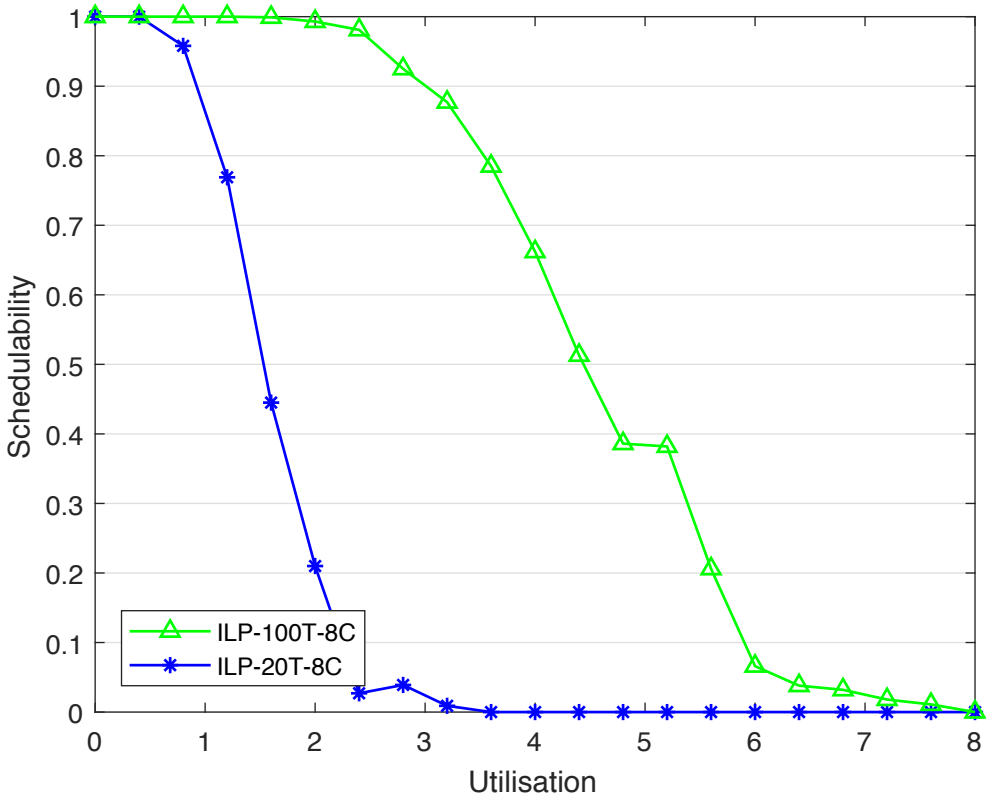


Figure 3.24: A plot illustrating the improved Schedulability on an 8 core platform when the number of tasks is also increased.

Experiment Four

Parameters:

- Multiple experimental runs were performed, each increasing the number of criticality levels by 1, from 2 to 5.
- 20 tasks were generated per task-set.
- Allocation was made to a 2 core platform.
- 4 minor cycles were included per major cycle.

Experiment four scales the number of criticality levels. The results of this experiment are shown in Figure 3.25. In this figure we observe the greatest weakness of WF, it scales poorly as the number of criticality levels is increased. A steady drop in schedulability may also be observed from ILP. A general decrease in schedulability is expected as for each criticality level an additional barrier (point of S^{max}) is required, which will cause some degree of schedulability loss.

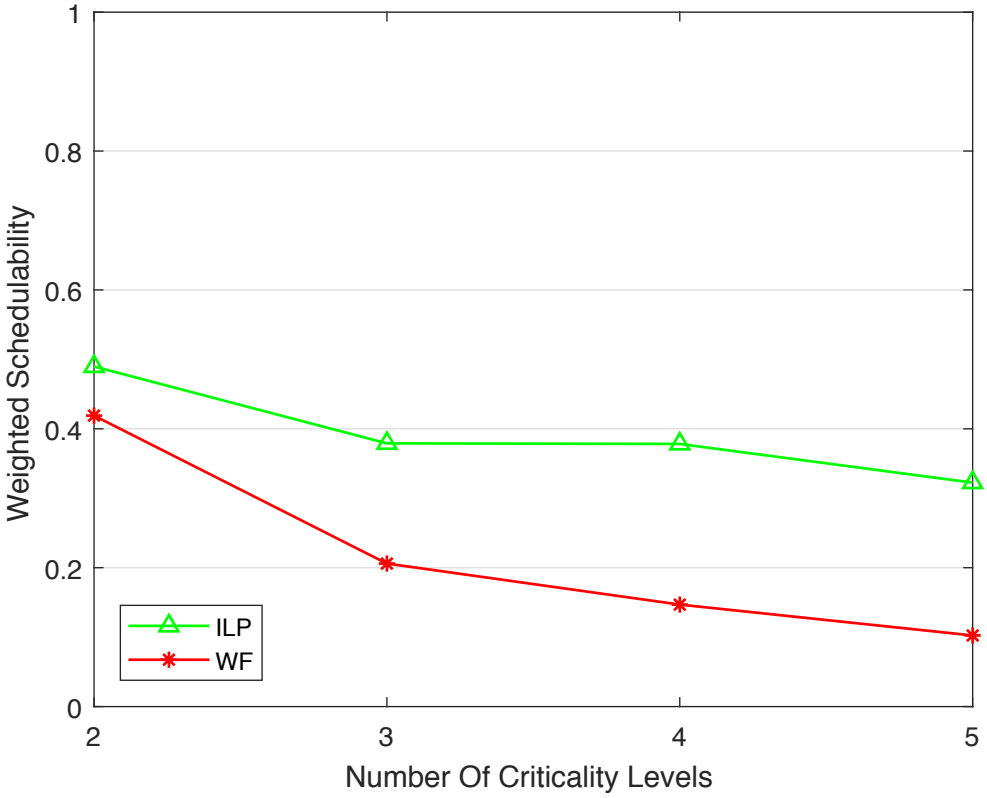


Figure 3.25: A plot illustrating the performance as the number of criticality levels is increased.

Experiment Five

Parameters:

- Multiple experimental runs were performed, each increasing the number of minor cycles from 4 to 16 in increments of 4.
- 20 tasks were generated per task-set.
- Allocation was made to a 2 core platform.
- Tasks were evenly distributed over two criticality levels.

Finally, we investigate the performance as the number of minor cycles is increased. The periods of tasks remain fixed at 25, 50 or 100. Thus we increase the number of minor cycles from 4 to 8, 12 and 16. The results are shown in Figure 3.26. The results here show little change across all techniques. This is likely to be due to the same periods being used for each number of minor cycles. Thus if a set is schedulable on 4 minor cycles, it is also schedulable on 8 if the largest period of any task is equal to the length of 4 minor cycles.

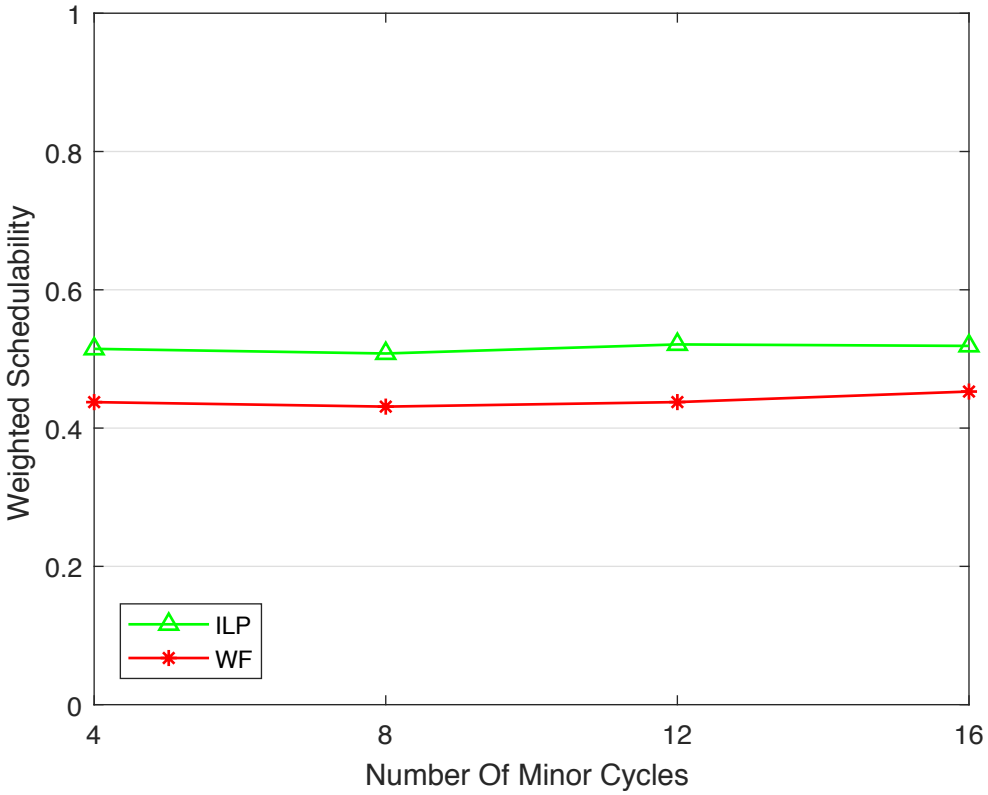


Figure 3.26: A plot illustrating the performance as the number of minor cycles is increased.

Timing

The schedulability results required further investigation. In short the question becomes: *How efficient is ILP in comparison to Worst Fit?* Timing data was taken for both the ILP solver and the WF implementation. While Gurobi does record the time taken to solve the model, we used Matlab's basic timing tools to take a reading for both the heuristic and the ILP solver to remain consistent. We took execution time data taken from the results shown in Experiment One is presented in Table 3.3:

	WF	ILP
Average	0.0058	0.0064
Median	0.0058	0.0017
Max	0.0185	58.618

Table 3.3: The average, median and max execution times of WF and ILP (in seconds).

The timing data in Table 3.3 illustrate that WF maintains a very low average, median and maximum execution times. All heuristic approaches allocate task sets within a fraction of a second. The ILP approach has an average execution time only slightly higher than the heuristic and a median execution time well below. A small number of outliers push the average execution time of ILP well above the median. However given than the median is very low it is clear that the vast majority of task sets are allocated very quickly within a fraction of a second. We observed the timing outliers at utilisations where task sets are borderline schedulable. While we included no timing cut-off for the ILP solver in Experiment One, we included a cut-off of 60 seconds for the scalability investigation. We observed only 0.3% of models generated reaching this limit over all of the timing data presented. This is a practical consideration as a large number of different experimental runs are required to scale all parameters. We accept that a number of outliers will occur but again observe a consistently low median execution time indicating that the vast majority of task sets are allocated very quickly.

We scaled the size of the task set, the number of cores, the number of criticality levels and the number of minor cycles and considered the execution time increase. We utilised box plots to present the resulting execution time data, the red centre line in each box is the median execution time, with the bottom and top whiskers being the 25th and 75 percentiles respectively. While a number of outliers exsist, this experiment illustrates that the majority of task sets are allocated very quickly. The data is presented in Figures 3.27, 3.28, 3.29 and 3.30 respectively.

It is clear from the average execution times and the box plot data that all approaches, no matter the scaling, take little time to execute for the vast majority of cases. However, we observe that the ILP based approach proves to be quicker to execute in all cases of comparable scale. As all timing values are taken using Matlab's in-built timing tools, the run-time recorded for ILP accounts for creation of the ILP model and its time running in the solver. The creation of the model is extensive but not exceedingly time consuming, it largely revolves around constructing a sparse matrix to represent the constraints.

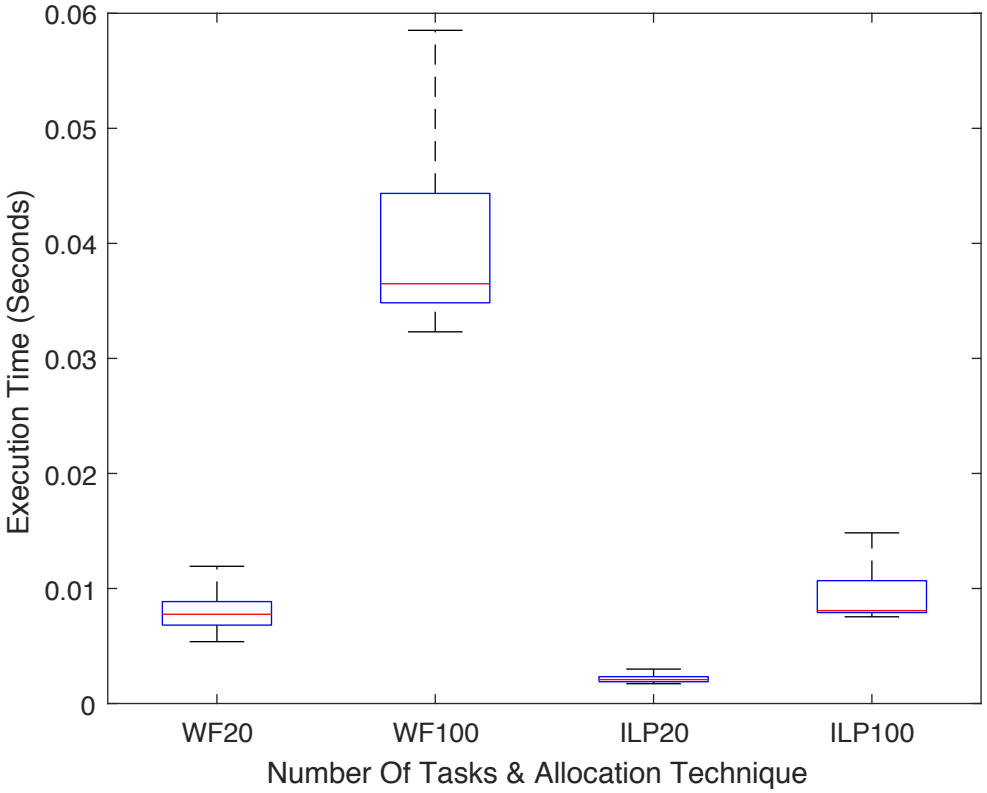


Figure 3.27: A box plot illustrating the range of execution times taken for each approach as the number of tasks is increased.

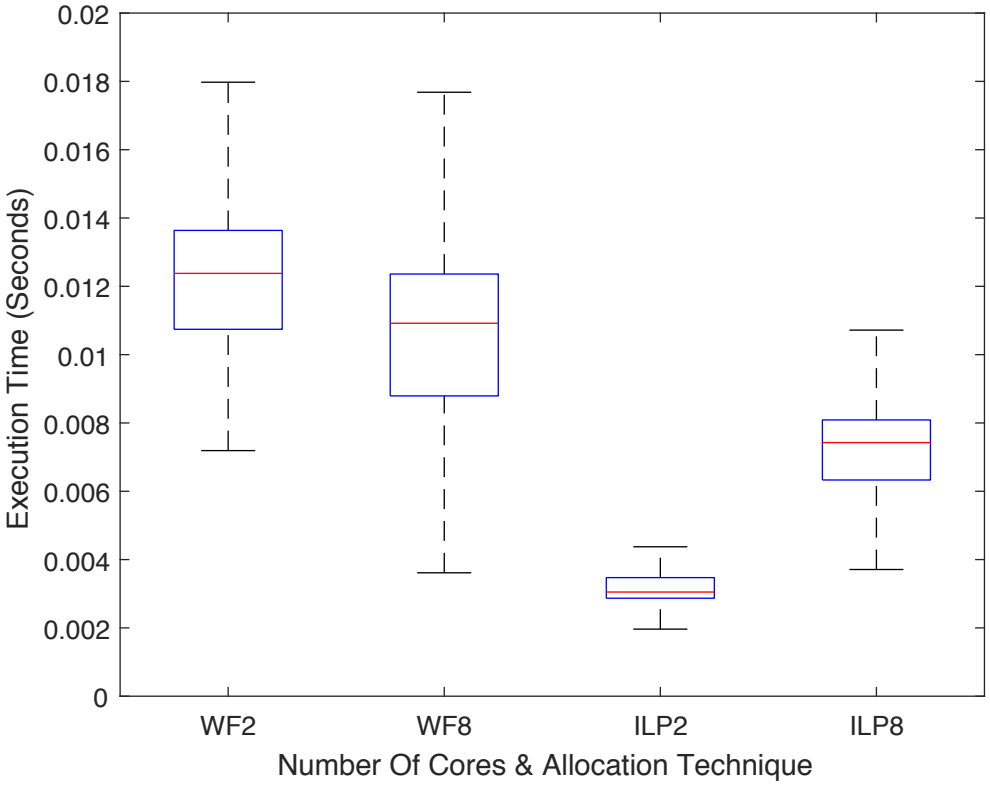


Figure 3.28: A box plot illustrating the range of execution times taken for each approach as the number of cores is increased.

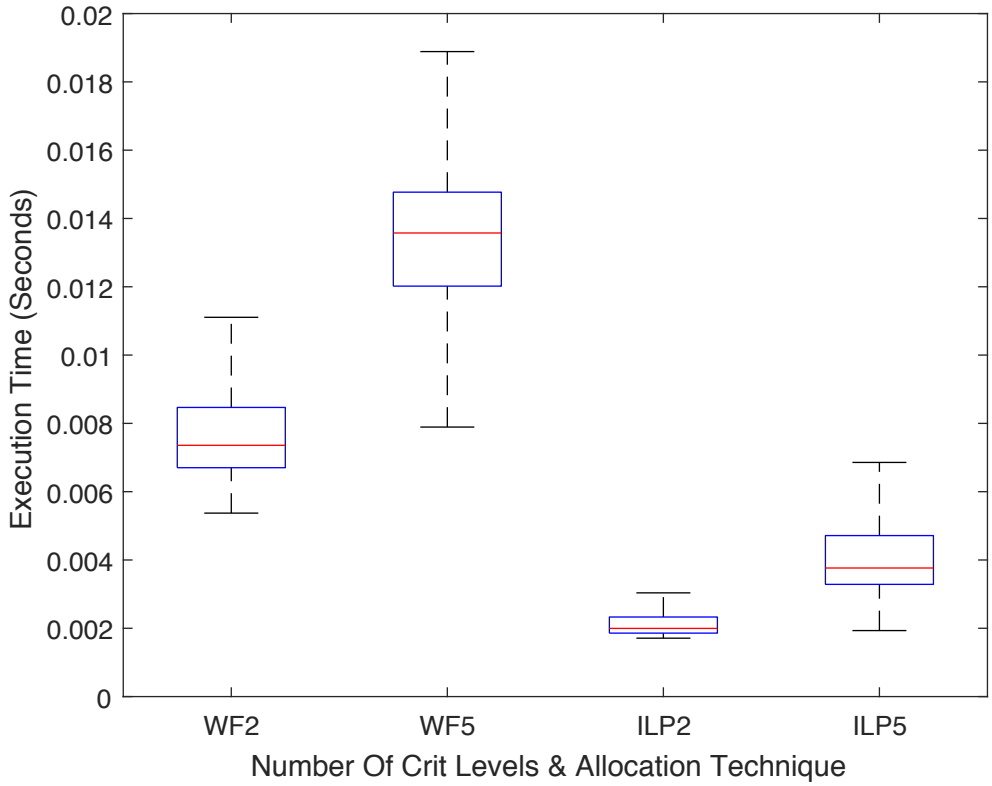


Figure 3.29: A box plot illustrating the range of execution times taken for each approach as the number of criticality levels is increased.

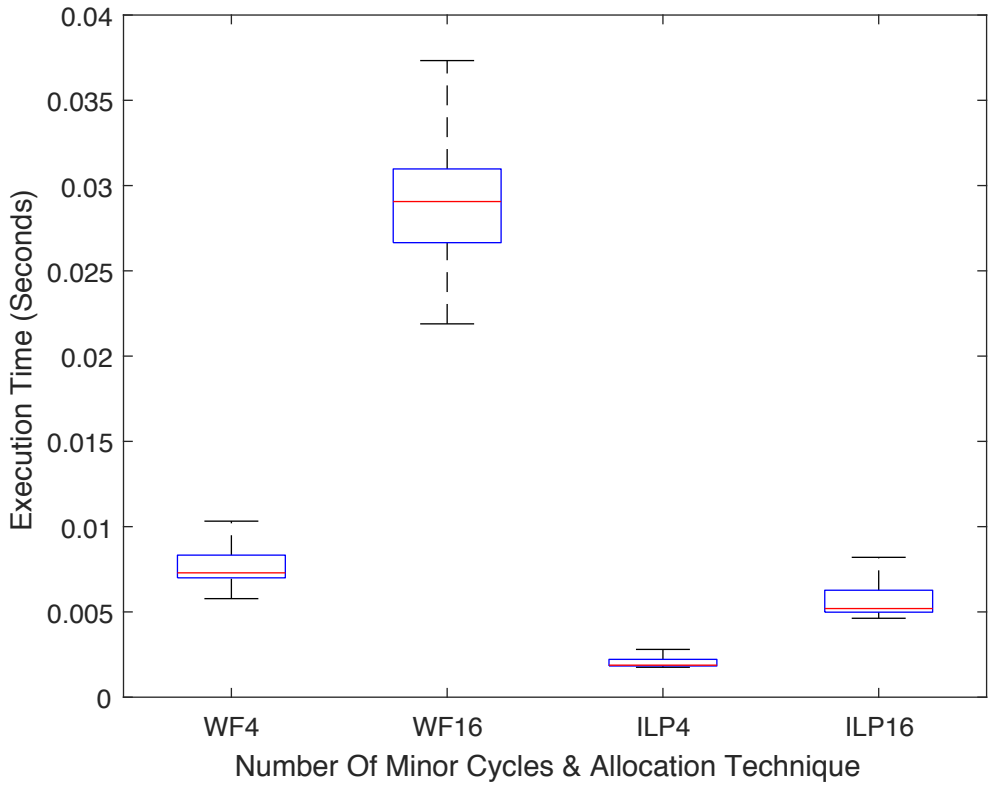


Figure 3.30: A box plot illustrating the range of execution times taken for each approach as the number of minor cycles is increased.

As our ILP models have no optimisation function, the solver is able to quickly find a solution and thus the overall runtime is very low (See the below 'Feasibility vs Optimisation' section for more insight). On the other hand, worst fit relies solely on a large number of iterations to find suitable allocations, as such its execution time is slightly larger than the ILP solution in this case.

We have shown that scaling the number of tasks, cores, criticality levels and minor cycles does have an impact on the execution time. However, the time taken to find feasible solutions remains extremely fast. In addition, given that this multiple minor cycle case is more complex than the single cycle, the results illustrating very fast performance also apply to the prior experiments in this chapter. Timing data was recorded for the experiment in Section 3.1.4, it shows extremely fast performance across all techniques, we included the details in Appendix A.

Summary

To conclude we summarise the results of the experiment, both the schedulability result and the timing result.

- Firstly: Our ILP implementation outperforms Worst Fit by a significant margin, particularly when scalability via the number of criticality levels is taken into account. This makes ILP an more effective candidate for allocation.

- Secondly: The timing data illustrates that, in this case our ILP approach actually out-performs WF in terms of execution time. This is likely down to implementation, ILP models are generated and run as feasibility tests using Gurobi[45], WF must undergo a large number of iterations in Matlab in order to find successful allocations. It may be possible to produce a faster implementation of worst fit, however both approaches are equally fast with a very small number of outliers.

While we do not claim that no better heuristic might exist, with the increase in schedulability offered by ILP and the very low execution times the logical choice is to make use of the optimal solver.

Feasibility vs Optimisation

They key reason for the efficient run-time of our ILP models, is their lack of an optimisation function. In effect these become feasibility tests rather than optimisations, the question becomes 'is there a search space?' rather than 'which point maximises a particular function?'. In these cases we are interested in constraint satisfaction rather than optimisation. The ability to generate models using tools such as Matlab makes ILP a compelling option when investigating the feasibility of a given task set on a mixed criticality cyclic executive. Model generation tools allow for the rapid development of system models which provide a system designer with a means of investigating possible task allocations during all stages of the design and development process.

Task Ordering With Criticality Level

During task allocation, tasks are allocated to the appropriate number of minor cycles, and within these to a core. Within this, each minor cycle on each core, is split via the barrier separating the execution of each criticality level. The ordering of execution within tasks of the same criticality level is not explicit, schedulability simply requires the tasks to fit in some order. As the barrier is a dynamic structure, even if a criticality change occurs, the barrier will still be invoked and some work of the following criticality level will execute. In a dual criticality context, logically, work scheduled later in the LO criticality execution is more likely to be effected by a criticality change (HI work overrunning the pre-computed point S). If a criticality change occurs, HI work overruns the pre-computed S point by some amount, once that work completes LO work begins execution. Work scheduled first in the LO mode is less likely to suffer during a criticality change. The notion of *Importance* could be used as a means of ordering tasks within a criticality level. Such an ordering could be applied to any criticality levels below the highest, with tasks assigned a higher importance executing earlier in the schedule. Ordering these tasks is simple, as the schedule is viable no matter which order the tasks execute.

In addition to simply ordering tasks once a schedule has been established, importance could be used as a factor for optimisation. The solver could be tasked with ensuring that where possible, higher importance tasks execute before lower

importance. This could either be modelled as firm constraints, where such an order is mandatory or as a desired order which might suffer if the only schedulable solution breaks it. It is worth noting that if modelled as firm constraints optimality is lost (in that it may not find a feasibility schedule, even if one exsists).We return to the issue of optimisation in Chapter 5.

3.3 Summary

In this chapter we have studied task allocation. We began at the single cycle level investigating how heuristics may be used to find correct schedules. Experimental results illustrated the effectiveness of the heuristics and the impact of the barrier separation technique. The use of ILP was considered as an optimal solver, its performance was compared against the heuristic techniques outperforming the heuristics by a small margin (aside from FF). We extended the model to consider multiple minor cycles and once again compared the heuristic against ILP. The results showed a significant improvement in schedulability with relatively low solution execution times.

In short, the main results of this chapter can be summarised as:

- Heuristic allocation techniques are effective for the single minor cycle allocation problem.
- Worst fit performs poorly in the multi-cycle scenario.
- ILP is effective at producing both single and multi-cycle schedules.
- ILP is efficient with very low runtime per solution.

This chapter illustrates that while for simpler cases heuristic allocation methods can be applied effectively, given the flexibility (in terms of increasing the problem size, more minor cycles, cores) and relatively low execution costs ILP is an excellent tool for mixed criticality cyclic executive task allocation.

Chapter 4

Task Splitting

One of the well documented drawbacks of Cyclic Executive platforms is their typically poor overall system utilisation. In the Mixed Criticality case this utilisation is hampered further by the introduction of a barrier protocol separating the execution of tasks with differing criticality levels. To address the issues surrounding utilisation this chapter investigates the use of task splitting to improve cyclic executive platform utilisation. The task splitting described is designed to be used in a limited way to help release any spare system utilisation.

Splitting in this chapter implies the suspension of execution after a given amount of time indicated by the length of the split. Splitting via pre-emption does introduce context switching overheads, a careful balance must be struck in order to find a gain in overall system utilisation. While we assume splitting via pre-emption, the work might also be considered in the context of physical code separation. This might create a situation where the system designer splits code manually to increase the available system utilisation on the advice of the solver.

When considering task splitting we make a key assumption. We assume that tasks may only be split across minor cycles and not CPU cores. This decision was made mainly to reduce the problem and ILP model complexity. In addition, cross core splitting requires suitable assurance that portions of the same task will not execute concurrently. Cross core splitting was dealt with in a mixed criticality cyclic executive context in [29]. While we do not claim that additional utilisation could not be gained by splitting across cores we focus on minor cycle splitting only.

This chapter will describe the splitting of mixed criticality tasks in two stages: the

first stage will discuss the splitting of LO criticality tasks and the second discusses the splitting of HI criticality tasks. While we restrict ourselves to two criticality levels, the observations and implementations apply to more than two levels, the LO splitting always applies to the lowest criticality and the HI splitting to all those criticality levels above the lowest.

4.1 Low Criticality Splitting

The splitting of the lowest criticality tasks is the simplest of the two stages. Conceptually, the act of splitting such tasks is simple, each task has only a $C_i(LO)$ to split, and thus may be split across minor cycles as desired. This is best clarified with an example task set and allocation. Consider the task set shown in Table 4.1

τ	$C(LO)$	$C(HI)$	T	L_i
τ_1	5	10	25	HI
τ_2	5	15	25	HI
τ_3	20	25	50	HI
τ_4	5	-	25	LO
τ_5	15	-	50	LO
τ_6	15	-	100	LO
τ_7	35	-	100	LO

Table 4.1: Task Splitting: An example task set to illustrate LO criticality splitting.

This task set assumes a 2 core system with 4 minor cycles where $T^F = 25$ and $T^M = 100$. It is clear that the $C_7(LO)$ of τ_7 is greater than the minor cycle length $C_7(LO) > T^F$, and thus for the system to be schedulable τ_7 must be split. A potential allocation for this task set is shown in Figure 4.1.

Each minor cycle is annotated as M_x where x is the minor cycle number, each core is indicated as C_y where y is the core number and each predicted barrier invocation is marked as $S^{MAX(m)}$. This allocation illustrates a potential split for τ_7 with 5 units of execution time in M_1, 20 in M_2 and 5 in M_3 and M_4. Splitting LO criticality tasks simply requires the solver to be provided with suitable variables and constraints, from there it will choose the task split.

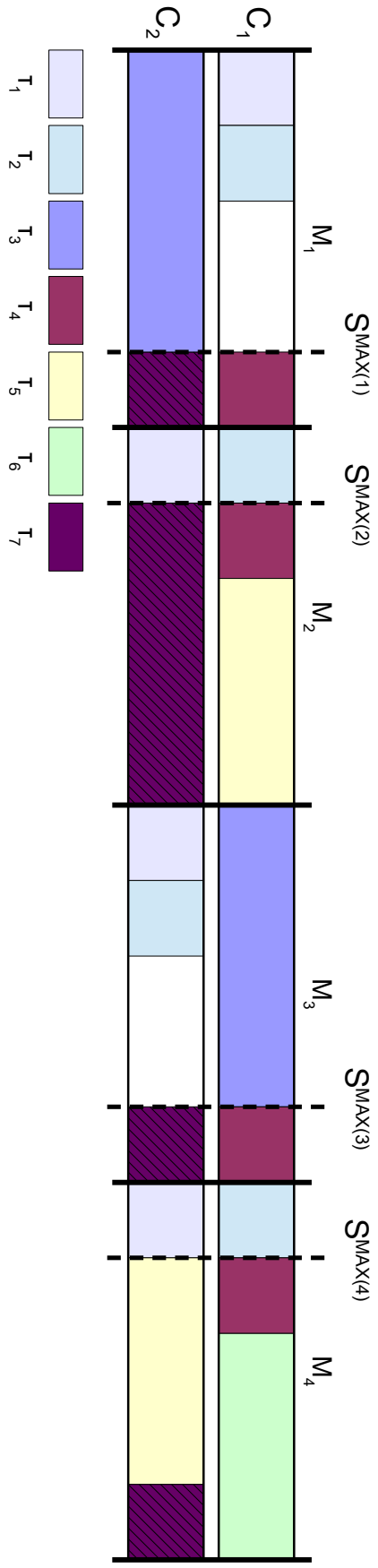


Figure 4.1: A schedule illustrating LO criticality splitting.

4.1.1 An MLP Model

While the splitting of LO tasks is conceptually simple, additional constraints and variables are required in our LP model to express the new allocation functionality. The most fundamental of these is the introduction of continuous variables, as such we move from Integer Linear Programming, to Mixed Linear Programming (MLP) (a combination of integer and continuous variables).

Continuous variables may be used to model task locations in a similar way to the binary variables we are already familiar with. If the continuous variable is constrained to be ≥ 0 and ≤ 1, then it may replace the binary variables for any LO criticality tasks that must be split. With a continuous variable, rather than requiring the task to be located in a single location, part of the task may be allocated to multiple locations. With these new variable types come some additional constraints to ensure the splitting performs as desired, we describe these below.

Minor Cycle Splitting

It is clear that if we constrain splitting to purely minor cycles, we require constraints to ensure that this is indeed the case. In order to achieve this we introduce a new set of binary variables, Y variables. For each split task there is a Y variable for each core. If a Y variable is set to 0, it indicates that no work is scheduled on that core. By constraining all the Y variables such that only one may equal 1, we ensure that tasks are split across minor cycles only. These constraints for a generic LO criticality task τ_z are as follows:

$$Q_z\_11 + Q_z\_12 + Q_z\_13 + Q_z\_14 - Y_1 = 0$$

$$Q_z\_21 + Q_z\_22 + Q_z\_23 + Q_z\_24 - Y_2 = 0$$

$$Y_1 + Y_2 = 1$$

The Y variables are included in the constraint alongside the task location variables. In order for the constraints to be met, either no work can be scheduled (all Q variables equal 0 and the Y variable equals 0) or all work exists (and a complete task is scheduled across a single core, with the Q variables adding up to 1), the Y variable equals 1 and thus the sum of the Q variables must equal 1 for the constraint

to satisfy its $= 0$ condition. The Y variables are then subject to a constraint which allows only one of them to equal 1, thus ensuring that a task is split only across minor cycles and not cores.

Thus following the same format, the variables for τ_7 are as follows:

Q7_11 + Q7_12 + Q7_13 + Q7_14 − Y1 = 0
Q7_21 + Q7_22 + Q7_23 + Q7_24 − Y2 = 0
Y1 + Y2 = 1

If an example allocation splits τ_7 in half with $Q7\_11 = 0.5$ and $Q7\_14 = 0.5$, the actual values of the constraint above would be.

$0.5 + 0 + 0 + 0.5 - 1 = 0$

$0 + 0 + 0 + 0 - 0 = 0$

$1 + 0 = 1$

It is clear that by allowing only a single Y variable to equal 1, we restrict a task to splitting across a single core. This constraint is repeated for each split task with a new Y variable defined for each core on every split task.

Maintaining Discrete Time

When introducing continuous variables we must account for the potential splits of WCETs produced. As the WCET constraints section multiplies the location variable(Q) by the WCET of the task, if that location variable is not properly constrained a task may result in providing a WCET which does not abide by integer units of time.

For example, if τ_7 had a split variable with the value 0.38, the result of the calculation to work out the WCET value schedule in that location would be, $35 \times 0.38 = 13.3$. The result is not permissible as WCETs must remain integer values.

In order to prevent this and constrain the model a constraint is required for each continuous variable, these are as follows:

For a generic LO criticality task τ_z:

$$C_z(LO) \times Q_z\_11 + J_1 = T^M$$
$$C_z(LO) \times Q_z\_21 + J_2 = T^M$$
$$C_z(LO) \times Q_z\_12 + J_3 = T^M$$
$$C_z(LO) \times Q_z\_22 + J_4 = T^M$$
$$C_z(LO) \times Q_z\_13 + J_5 = T^M$$
$$C_z(LO) \times Q_z\_23 + J_6 = T^M$$
$$C_z(LO) \times Q_z\_14 + J_7 = T^M$$
$$C_z(LO) \times Q_z\_24 + J_8 = T^M$$

The J_x variables are integers with an upper bound of T^M (although in reality this just needs to be suitably large, T^M is not required). By adding the split WCET (the location variable multiplied by the WCET) with an integer variable and requiring the result of the constraint to be integer, we ensure that any split task does not violate the notion of discrete units of time.

For example, given τ_7 (from Table 4.1), we established above that if one of its variables, say Q_7_11 (core 1, minor cycle 1) contained the value 0.38 the resulting multiplication with its WCET would not result in an integer(the actual result is 13.3). Thus if that split is constrained as follows it would not satisfy the constraint.

$$C_7(LO) \times Q_7\_11 + J_1 = T^M$$

Or

$$35 \times 0.38 + J_1 = 100$$

As J_1 is an integer there is no value it can take to satisfy the constraint. However, if $Q_7\_11 = 0.4$ then $35 \times 0.4 = 14$ thus $J_1 = 86$:

$$35 \times 0.4 + 86 = 100$$

Now the resulting split results in an integer WCET the constraint may be satisfied.

Given that J_1 is an integer constrained $0 \leq J_1 \leq T^M$, the result of $C_7(LO) \times Q_7\_11$ must also result in an integer for the constraint to hold. Every variable is constrained in this way where splitting is permitted.

For τ_7:

20 Q7_11 + C1 = 100
20 Q7_21 + C2 = 100
20 Q7_12 + C3 = 100
20 Q7_22 + C4 = 100
20 Q7_13 + C5 = 100
20 Q7_23 + C6 = 100
20 Q7_14 + C7 = 100
20 Q7_24 + C8 = 100

Clearly in addition to these constraints, any variables which have changed from binary to continuous must be declared as such at the bottom of the model. With these two additional sets of constraints in mind the model overview can be updated and is shown again in Figure 4.2.

4.1.2 Splitting when required

Our MLP model at this stage maintains a desirable property: regardless of which tasks are permitted to split, if an ILP (non-splitting) solution exists, the model will find it. This allows a system designer to specify a number of tasks for splitting while being sure that if splitting is not required, the integer solution will be found.

To this point, our ILP and MLP models all lack an objective function. While the location of an objective function is included in all model overviews (such as Figure 4.2), this section of the model remains empty. This causes the LP solver to seek the initial feasible solution, rather than optimise some value. Essentially, the solver simply seeks to determine if a search space exists. The fact that our LP models are, to this point, feasibility tests has important repercussions.

- The very fast performance of our ILP models can be largely attributed to be-

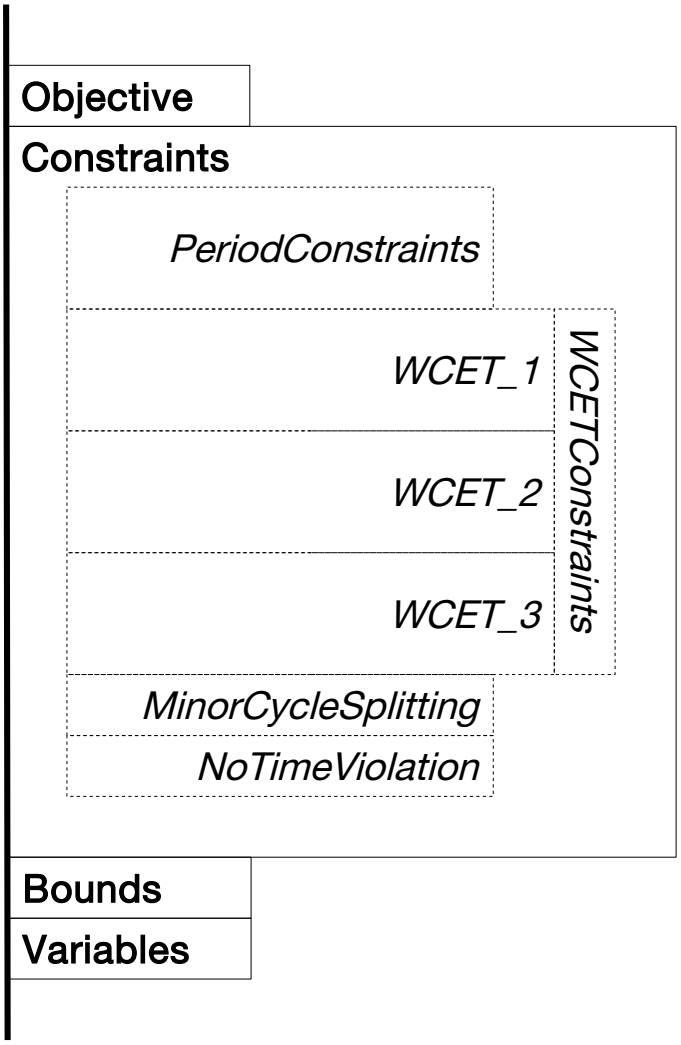


Figure 4.2: Abstract Diagram: Full (Splitting)

ing feasibility tests rather than optimisations. The solver will halt on discovery of the initial feasible solution (e.g. it confirms a search space exists).

- Ceasing at the initial feasible solution explains why our MLP models produce an ILP solution where possible. As our solver (Gurobi [45]) seeks an initial feasible solution via the simplex algorithm [32], the variables are initiated using the bounds placed upon the constraints. Therefore, if the constraint is bound with an integer value, the initial values the solver attempts to assign to each variable will be that same integer. In this way our MLP models split only when non of the initial integer assignments are valid.

The following section discusses how HI criticality tasks may be split. Detailed evaluations of both LO and HI criticality splitting are presented in Section 4.3 at the end of the chapter.

4.2 HI Criticality Splitting

The splitting of tasks where the criticality level is greater than the lowest becomes more complicated. This complexity is due to a task being assigned multiple WCET predictions (in this work, two WCETs), thus we must consider how the splitting of tasks is handled during all criticality modes. While this applies to any criticality level greater than the lowest, we stick to a dual criticality system (LO and HI where $LO < HI$) and thus consider the splitting of tasks in the HI criticality mode.

4.2.1 Period Transformation

When considering how HI criticality splitting is handled we return to the work of Vestal [81] and its follow-ups [38][1]. We have covered Period Transformation in Chapter 2.2.3, however we revisit it here highlighting the key features relevant to this section. Period Transformation was used in this context to allow any HI criticality tasks with periods greater than the shortest period of any LO criticality task to be transformed. In this way a criticality monotonic ordering could be created removing any chance of criticality inversion (where a LO criticality task pre-empts a HI). To do this tasks were transformed by some factor termed m:

$$m = \frac{T_i}{T_l}$$

Where T_i is the HI criticality task being transformed and T_l is the LO criticality task with the shortest period.

Crucially, [38][2] found that a split HI task must execute in segments of $C_i(HI)/m$ until it reaches its untransformed $C_i(LO)$ value. This is because it is only at this point that the system is able to determine whether a task will overrun and a criticality change is needed. Up to this point, execution must be performed in such a way that if an overrun occurs, the maximum execution time can be provided. While this approach is not directly applicable in the Cyclic Executive context as we consider only the splitting of WCETs (periods are fixed), it provides the inspiration for the following approach.

[1] While the work was undertaken by this author it was part of an earlier degree [37]
[2] Work of this author as part of a prior degree

4.2.2 Containers

In order to split HI criticality tasks we reconsider the structure of a task with multiple execution times. Up to this point we considered a HI criticality task, τ_i, with two execution times, one for the LO criticality $C_i(LO)$ and one for the HI $C_i(HI)$. Instead such a task might be seen to have again a LO criticality value $C_i(LO)$ but in addition a value $C_i(EX)$ which represents the additional execution time required to reach a full $C_i(HI)$ value. In other words:

$$C_i(EX) = C_i(HI) - C_i(LO)$$

When we define our models with HI criticality task τ_i permitted to split, we create containers for $Ct_i(LO)$ & $Ct_i(EX)$. These containers are constrained in such a way that we allow the solver to allocate additional work to the LO container and thus for execution in the LO criticality mode if desired, this balance is decided by the solver. For clarity the containers must be constrained as follows:

- $C_i(LO) \leq Ct_i(LO) \leq C_i(HI)$
- $0 \leq Ct_i(EX) \leq (C_i(HI) - C_i(LO))$
- $Ct_i(LO) + Ct_i(EX) = C_i(HI)$

Once a distribution between the two containers is allocated, the task may then be split as desired. $Ct_i(LO)$ may be split into any number of minor cycles, however $Ct_i(EX)$, or split instances of, must be allocated only to the final minor cycle to which $Ct_i(LO)$ is allocated and any after. This is logical as the EX execution, or the HI criticality mode execution, only may occur once the LO work has completed. The process of allocation is described below:

- The LO container, $Ct_i(LO)$, is allocated execution time of the task such that $C_i(LO) \leq Ct_i(LO) \leq C_i(HI)$.
- The EX container, $Ct_i(EX)$, is allocated the remaining execution time for the HI criticality mode. $Ct_i(EX) = C_i(HI) - Ct_i(LO)$.

  The task is now represented such that $Ct_i(LO) + Ct_i(EX) = C_i(HI)$.

- $Ct_i(LO)$ may now be split across a number of minor cycles, each split instance of $Ct_i(LO)$ is represented by $Ct_{ij}(LO)$ where j is the minor cycle.

$$Ct_i(LO) = \sum_{j \in T^M} Ct_{ij}(LO)$$

Each instance of $Ct_{ij}(LO)$ may be of differing lengths, such an allocation is illustrated in Figure 4.3.

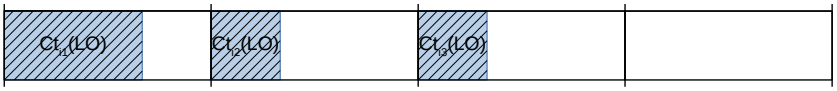


Figure 4.3: A split LO container.

- Following the allocation of $Ct_i(LO)$, $Ct_i(EX)$ must be allocated. Each instance is again described as $Ct_{ij}(EX)$ where j is the minor cycle.

$$Ct_i(EX) = \sum_{j \in T^M} Ct_{ij}(EX)$$

Split instances of $Ct_i(EX)$ may only be allocated the minor cycle, and all those following the final allocation of $Ct_i(LO)$. Figure 4.4 displays the final allocation.

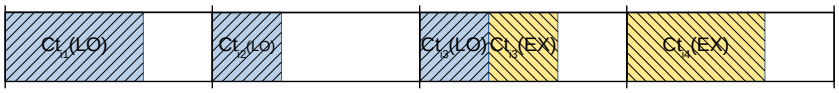


Figure 4.4: Split LO and EX containers.

While it is unlikely that without an optimisation, work will be moved from $Ct_i(EX)$ to $Ct_i(LO)$ the capability is there. The reasoning behind this ability is that if more work is considered LO, then there will be a reduced chance of a criticality change occurring which would affect the entire system.

To illustrate, consider the task set in Table 4.2, an extension of Table 4.1 which includes an additional task, τ_8 and τ_7 has a lower WCET value of 20.

We now consider the scenario where τ_3 is allowed to split. A potential allocation of this task set is shown in Figure 4.5. This scenario illustrates how τ_3 is split over minor cycles 1 and 2, but is not split over cycles 3 and 4. The solver only splits tasks where it is required.

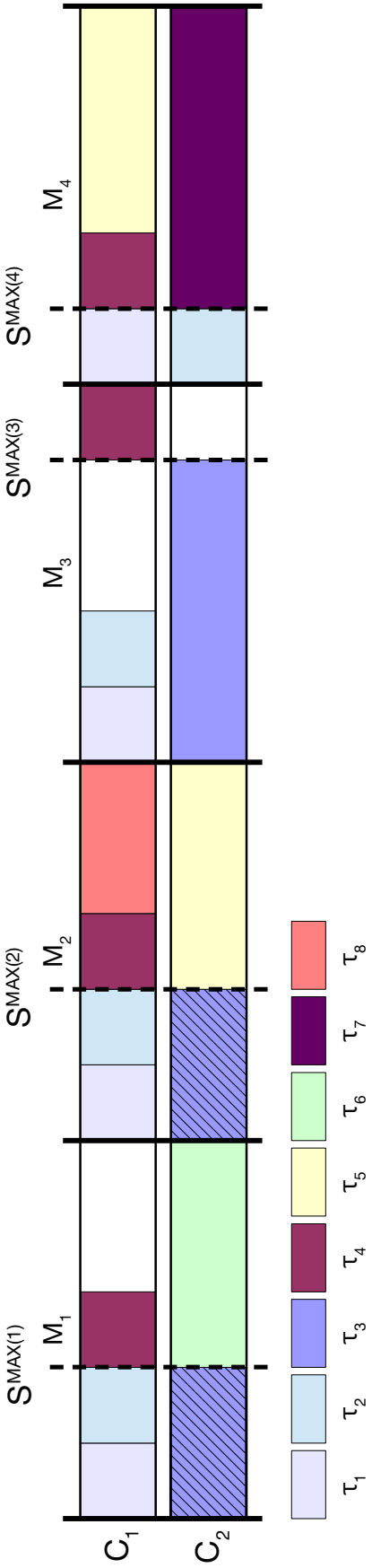


Figure 4.5: A schedule illustrating HI criticality splitting.

τ	$C(LO)$	$C(HI)$	T	L_i
τ_1	5	10	25	HI
τ_2	5	15	25	HI
τ_3	20	25	50	HI
τ_4	5	-	25	LO
τ_5	15	-	50	LO
τ_6	15	-	100	LO
τ_7	20	-	100	LO
τ_8	10	-	100	LO

Table 4.2: Task Splitting: An example task set to illustrate HI criticality splitting.

4.2.3 Updated MLP model

As with splitting LO criticality tasks, a number of additional constraints are required for the MLP model to include the new features. In addition, the model must be adapted to suit the new container based approach.

Periodicity Constraints

Previously we made use of separated LO WCET and HI WCET values for each task in the system. These values are multiplied by binary variables in the case of ILP and potentially continuous variables in the MLP models for LO criticality task splitting. However, given the new situation, where work may be migrated into the LO container if required, we must calculate the WCET relevant for each container. This begins with the initial set of periodicity constraints, if τ_i is a HI criticality task, in the situation where splitting is not required the constraints would be as follows[3]:

$$Q_i\_11 + Q_i\_21 + Q_i\_12 + Q_i\_22 = 1$$

$$Q_i\_13 + Q_i\_23 + Q_i\_14 + Q_i\_24 = 1$$

Whereas the statement differs significantly if the task must be split:

[3]for a system for 2 cores and 4 minor cycles where τ_i has $T_i = T^M/2$

$$Q_i\_11 + Q_i\_21 + Q_i\_12 + Q_i\_22+$$
$$Q_i\_11EX + Q_i\_21EX + Q_i\_12EX + Q_i\_22EX = n$$
$$Q_i\_13 + Q_i\_23 + Q_i\_14 + Q_i\_24+$$
$$Q_i\_13EX + Q_i\_23EX + Q_i\_14EX + Q_i\_24EX = n$$

Firstly, it is clear that the standard variables, Q_i_11, represent $Ct_i(LO)$ and additional variables, Q_i_11EX, are introduced to represent $Ct_i(EX)$. In addition, it follows that as the whole task is represented by these variables, the sum of all must be used to enforce periodicity constraints. Finally, rather than requiring that the variables be equal to 1, eg a whole task, we require them to be equal to n. The value n is found by $n = C(HI)/C(LO)$, rather than a task's components being equal to 1, or having a split portion of 1, they are equal to or contain a split portion of n. Thus only a single LO WCET must be fed into the model in order to calculate what portion of the WCET of the task is assigned to a location, the HI WCET is represented by $n \times C(LO)$. To illustrate this, if τ_3 from Table 4.2 must be split, the constraints would be as follows:

```
Q3_11 + Q3_21 + Q3_12 + Q3_22 + Q3_11EX + Q3_21EX
+ Q3_12EX + Q3_22EX = 1.25
Q3_13 + Q3_23 + Q3_14 + Q3_24 + Q3_13EX + Q3_23EX
+ Q3_14EX + Q3_24EX = 1.25
```

As $C_3(LO) = 20$ and $C_3(HI) = 25$, $n = 25/20$, thus an entire task is scheduled by including 1.25 in some way across the given variables.

An additional constraint is required to ensure that $Ct_i(LO) \geq C_i(LO)$ holds, this constraint reads:

$$Q_i\_11 + Q_i\_21 + Q_i\_12 + Q_i\_22 \geq 1$$
$$Q_i\_13 + Q_i\_23 + Q_i\_14 + Q_i\_24 \geq 1$$

And for τ_3:

Q3_11 + Q3_21 + Q3_12 + Q3_22 >= 1

Q3_13 + Q3_23 + Q3_14 + Q3_24 >= 1

These are included as part of the periodicity constraints as they dictate, where and how frequently a task may be allocated.

WCET Constraints

As we have both introduced new variables and altered the way split HI tasks calculate the WCET of splits we must update the WCET constraints section to illustrate the new functionality. The WCET constraints read as follows and is repeated for all combos of c and m.

$$\forall i, \sum_{i \in hctS} C_i(LO) \times Q_i\_cm + C_i(LO) \times Q_i\_cmEX + \sum_{i \in hct} C_i(HI) \times Q_i\_cm \leq T^F$$

Where $hctS$ is the set of high criticality tasks that are permitted to split and hct is the set of non-splitting high criticality tasks.

Given 3 HI criticality tasks, (τ_i, τ_l,τ_j), τ_i is allowed to split, the WCET calculation for the HI mode (WCET 1) in core 1 minor cycle 1 ($c = 1, m = 1$):

$$C_i(LO) \times T_i\_11 + C_i(LO) \times T_i\_11EX + C_l(HI) \times T_l\_11 + C_j(HI) \times T_j\_11 \leq T^F$$

The example shows how both the $Ct_i(LO)$ and $Ct_i(EX)$ containers are included, only multiplying both by $C_i(LO)$ as the sum of each set of variables are constrained above to be equal to n based on their periodicity. The example constraint for τ_1, τ_2 & τ_3 from Table 4.2 where τ_3 is allowed to split is shown below for core 1 minor cycle 1:

10 Q1_11 + 15 Q2_11 + 20 Q3_11 + 20 Q3_11EX <= 25

In the second set of WCET constraints (WCET_2) only $Ct(LO)$ values are included as they represent the LO criticality execution of the task. The WCET_2 and WCET_3 constraints remain the same as those presented in Chapter 3, Section 3.2.

Minor Cycle Splitting

The constraints which control splitting to ensure it is over minor cycles only need minor adjustments to account for the new functionality. We use $10 \times Y$ rather than just Y as a binary variable, as $n > 1$, and the value provided by the Y variable must be able to satisfy the constraint on its own. The constraint is also required to be greater than or equal to n, thus either the location variables contain a complete split task, or the Y variable is used to satisfy the constraint. The constraints for HI task τ_i are as follows:

$$Q_i\_11 + Q_i\_12 + Q_i\_11EX + Q_i\_12EX + 10 \times Y_1 \geq m$$
$$Q_i\_21 + Q_i\_22 + Q_i\_21EX + Q_i\_22EX + 10 \times Y_2 \geq m$$
$$Q_i\_13 + Q_i\_14 + Q_i\_13EX + Q_i\_14EX + 10 \times Y_3 \geq m$$
$$Q_i\_23 + Q_i\_24 + Q_i\_23EX + Q_i\_24EX + 10 \times Y_4 \geq m$$
$$10 \times Y_1 + 10 \times Y_2 = 10$$
$$10 \times Y_3 + 10 \times Y_4 = 10$$

And thus, for τ_3 the constraints read:

```
Q3_11 + Q3_12 + Q3_11EX + Q3_12EX + 10 Y1 >= 1.25
Q3_21 + Q3_22 + Q3_21EX + Q3_22EX + 10 Y2 >= 1.25
Q3_13 + Q3_14 + Q3_13EX + Q3_14EX + 10 Y3 >= 1.25
Q3_23 + Q3_24 + Q3_23EX + Q3_24EX + 10 Y4 >= 1.25
10 Y1 + 10 Y2 = 10
10 Y3 + 10 Y4 = 10
```

As τ_3 has a period $T_3 = 50$ this constraint ensures that the task is allocated across the same core during minor cycle 1 and 2 and the same core across minor cycle 3 and 4. One of the first two Y variables, $Y1$ & $Y2$, must equal 10, thus requiring no work be allocated across a particular core during the interval they represent.

No Time Violations

Finally, this section requires only a minor update to ensure all EX variables are also checked to ensure that no integer time violations are permitted. The EX variables are simply included alongside the standard variables as illustrated in Section 4.1.1.

4.3 Experiment: MLP Schedulability Gains from Task Splitting

These experiments are designed to assess the improvements in schedulability gained from limited task splitting. We investigate how schedulability is related to the number of tasks split in both the LO and HI modes. The scalability of the splitting approach is considered by varying the number of tasks, cores, criticality levels and minor cycles. Finally we consider how including limited splitting and the change to MLP models affects the execution time.

Setup

- We Generated 1000 task sets per 5% increase in utilisation.
- Task utilisations were generated using UUniFast, an algorithm presented in [25] which provides an unbiased distribution of utilisation values, following standard practice in synthetic task set generation.
- The minor cycle length was set at 25, with the major cycle length set at 100 ($T^F = 25, T^M = 100$)
- Task periods were selected at random from either 25, 50 or 100.
- Deadlines were set equal to periods. $D_i = T_i$.
- The LO execution times of each task were produced as follows: $C_i(LO) = U_i/T_i$

- For tasks with a criticality greater than the lowest, their HI execution times were determined by $C_i(L_i) = C_i(LO) * CF$ - CF is the criticality factor, a random value between 1.2 and 2.
- Timing data was recorded to find the average time each approach took to find a solution. All timing data was recorded on a 4 core Intel i7 4790K.
- The barrier protocol was implemented for all allocation techniques.
- Only tasks of periods 50 or 100 were split (as splitting is only permitted over minor cycles).

Results

Experiment One

Parameters:

- 20 tasks were generated per task-set.
- Allocation was made to a 2 core platform.
- Tasks were evenly distributed over two criticality levels.
- 4 minor cycles were included per major cycle.

Experiment one consists of a number of plots investigating the impact of splitting different numbers of tasks. We begin by investigating the impact in schedulability of splitting in the extreme case where all tasks which are able to split (those with periods greater than 25) are split. The fully split approach is compared with a standard (no splitting) ILP allocation. The results are shown in Figure 4.6. Somewhat unsurprisingly this plot illustrates a significant increase in schedulability if all possible tasks are split. The main result from this experiment is to illustrate the maximum potential schedulability gain which can be provided by splitting. In reality, allowing all tasks to split would result in the solver splitting tasks in very unpredictable ways, particularly as the utilisation of task sets increases. From our observations this creates a large number of split tasks, which may be counter productive if context switching costs were introduced.

With this maximum in mind we constructed an experiment investigating the schedulability gain in splitting all possible LO criticality tasks and all possible HI criticality tasks. Again we compare the result with a standard ILP allocation. The results are shown in Figure 4.7:

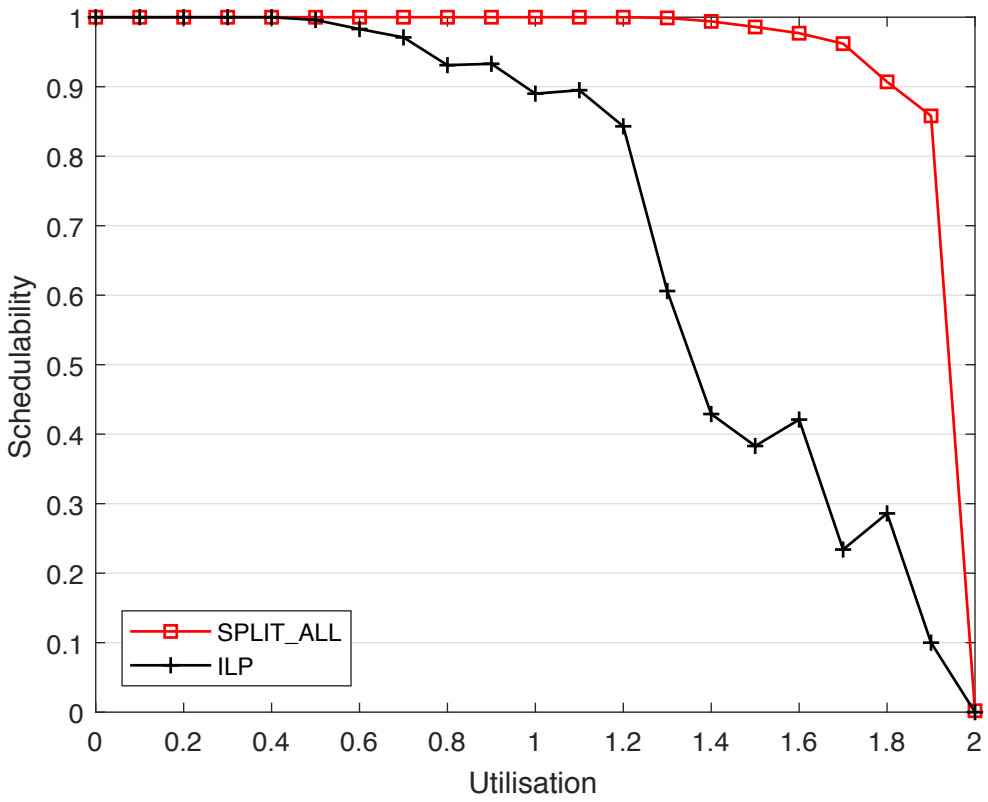


Figure 4.6: A graph illustrating the performance in schedulability where all and no tasks are split.

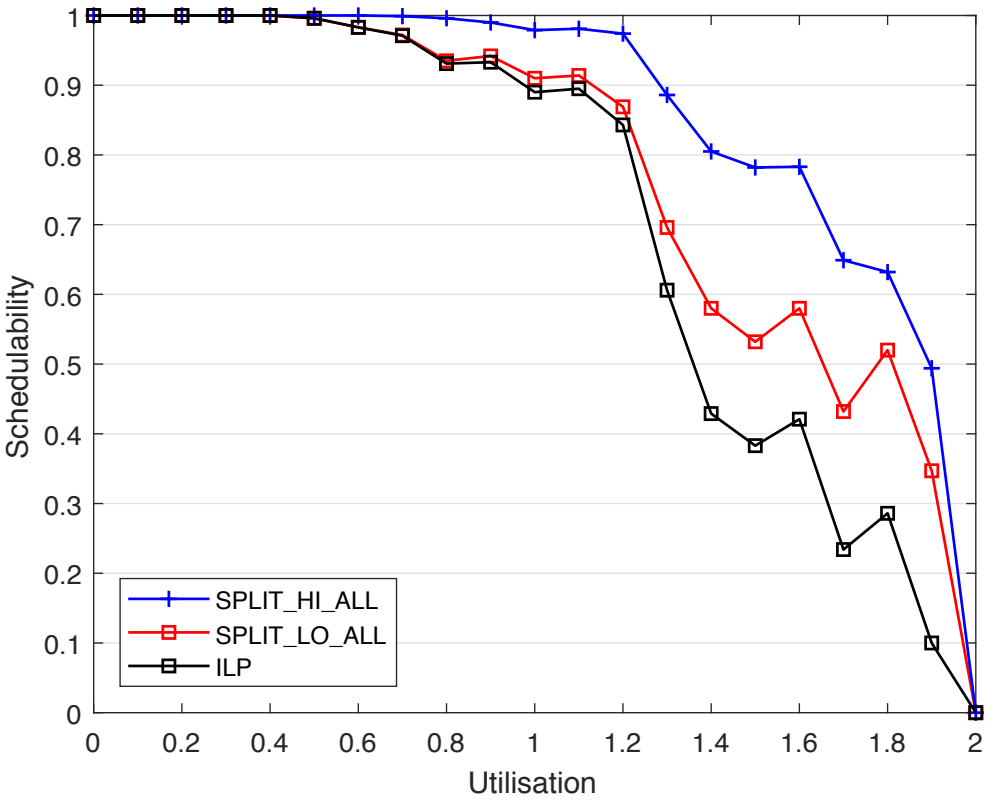


Figure 4.7: A graph illustrating the performance in schedulability where all LO, HI and no tasks are split.

Here we see a schedulability gain when splitting both the LO and HI criticality tasks. While the plot which splits all HI criticality tasks performs significantly better than the LO splitting and ILP plots, these lines will be subject to change based on the parameters of the task generation. The balance of LO and HI criticality tasks alongside their relative utilisations in the task set will affect these results. Regardless, any degree of splitting appears to have a significant impact on schedulability.

To investigate further we compare the approaches splitting all possible LO and HI tasks, with approaches just splitting the LO criticality task with the largest WCET and the HI criticality task with the largest WCET. The results are illustrated in Figure 4.8. In the LO criticality case the model splitting just a single task performs almost as well as the model which splits all of them. Meanwhile, the HI criticality model splitting the HI criticality task with the largest WCET performs very well, improves schedulability greatly and comes close to the the model where all HI criticality tasks are split. This illustrates a very interesting feature, splitting only a single task can have a large impact on the overall schedulability. Thus our notion of limited splitting has merit, as splitting only a small number of tasks can have a large impact on schedulability. From these results, the fact that the single split tasks are those with the largest WCETs and our own observations it becomes clear that often simply splitting a single task effectively *unblocks* the allocation problem. The freedom to split just a single high WCET task has a large impact on overall schedulability as it seems to give the solver many more options when seeking a suitable allocation.

Finally, for a somewhat more realistic example, we compare an approach which permits tasks to split only if their WCET is greater than a certain threshold. In this case the threshold is 20 (as $T^F = 25$, the actual values are multiplied to help maintain integer time units), the results are presented in Figure 4.9. The resulting plot illustrates that such an approach to splitting can have a significant impact in the overall schedulability. Of course, the threshold of 20 is used here simply as an example, any value could be implemented depending upon the requirements of the system. Overall, we wish to illustrate that splitting can be controlled in some smart way such that it is only permitted for a small number of tasks with extreme parameters.

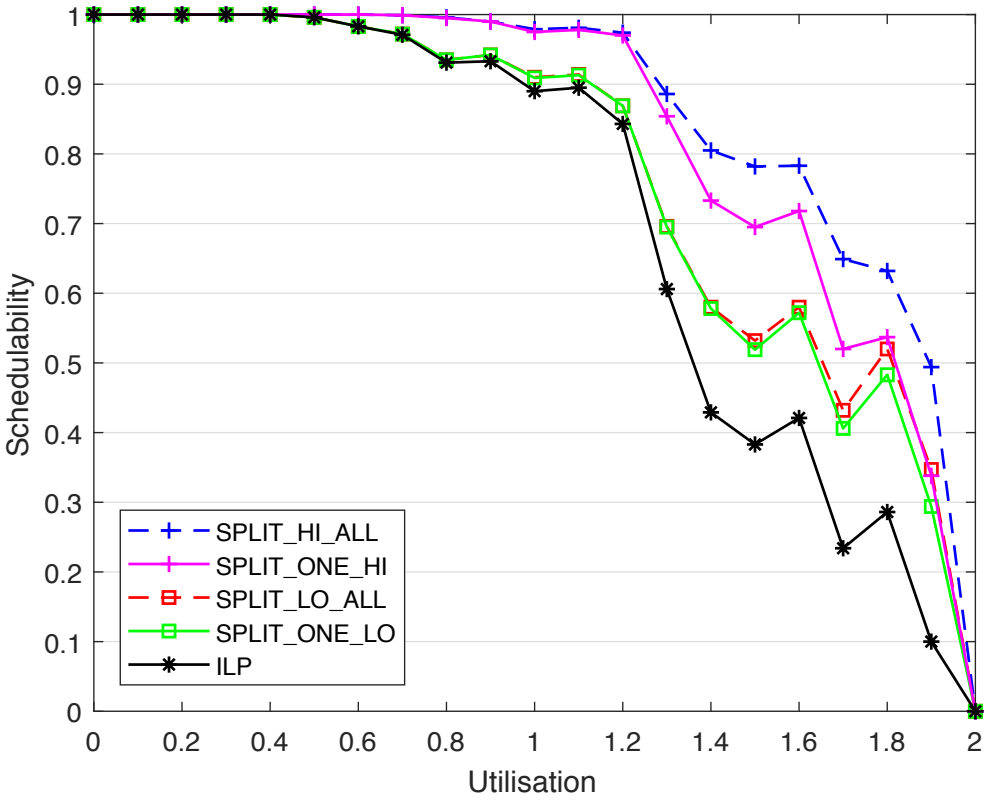


Figure 4.8: A graph illustrating the impact of splitting just a single task compared to splitting all within a criticality level.

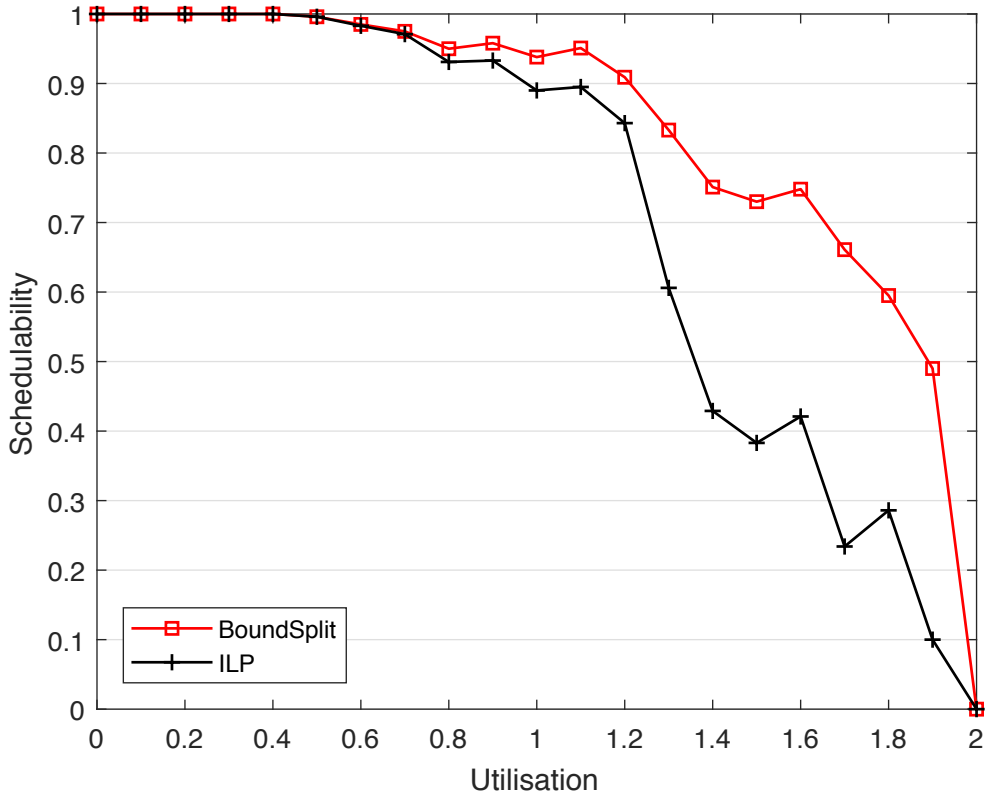


Figure 4.9: A graph illustrating the impact of splitting only those tasks with WCETs over a certain threshold.

Experiment Two

Parameters:

- Multiple experimental runs were performed, each increasing the number of tasks per set in increments of 20, from 20 to 100.
- Allocation was made to a 2 core platform.
- Tasks were evenly distributed over two criticality levels.
- 4 minor cycles were included per major cycle.

As we have done in previous experiments, we investigated the scalability of the approach. To achieve this we continued investigating the approach where all possible tasks are split compared to ILP. We chose the *SplitAll* approach over any of the single split alternatives as it illustrates the extreme case vs absolutely zero splitting. It is likely that the results of these experiments are directly relevant to approaches splitting a small number of tasks, with the same scalability results holding true while the schedulability would be reduced by some degree relative to what we saw in Experiment One.

The results are shown in Figure 4.10. As the number of tasks per set increases so does the schedulability. This is due to an increased granularity in allocation giving the solver more options to allocate around the synchronised criticality switching reducing wasted utilisation. Curiously, it can be noted here that the ILP approach improves schedulability quickly as the number of tasks per set is increased when compared to the *SplitAll* approach. While ILP does not reach the same level of performance, it may be the case that for systems with a very large number of tasks, splitting may not have as much benefit. However, these are synthetic task sets, as such as the number of tasks increases, the chance of any one task having a particularly large WCET decreases, and thus the advantage gained from task splitting is reduced. As this experiment limits the number of cores to 2, the size of each of the 100 tasks is likely to be very small. A *real world* system may have a large number of tasks and include some tasks with very large WCETs, in this case task splitting may well be beneficial.

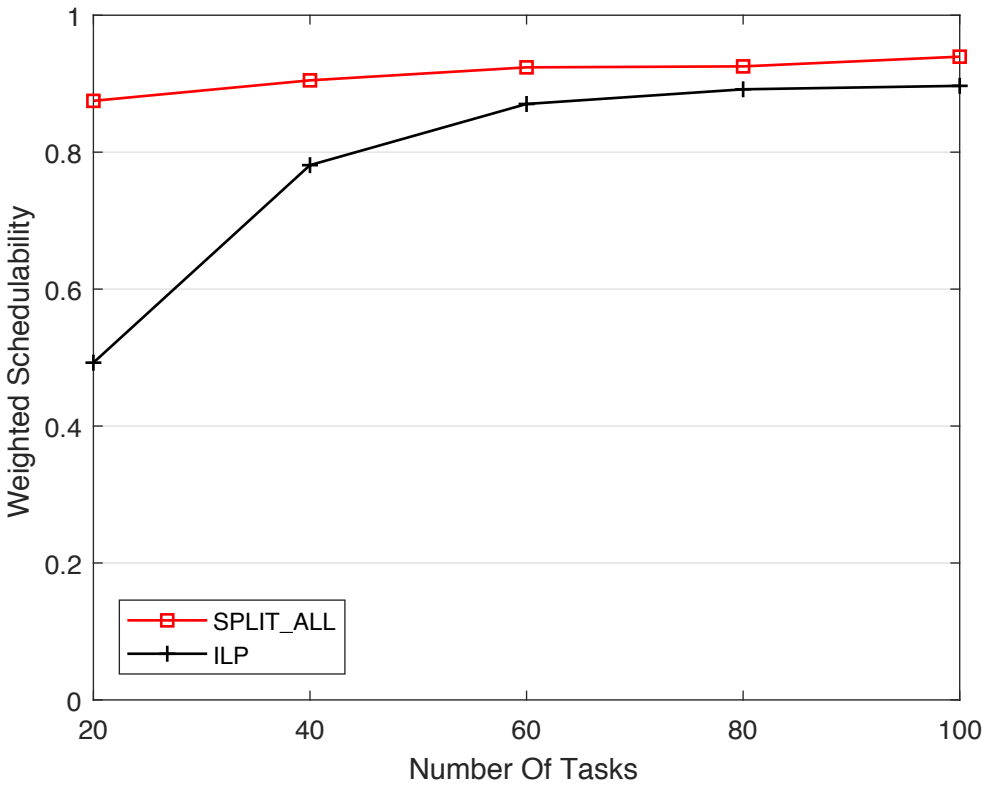


Figure 4.10: A graph utilising weighted schedulability as a metric to determine the performance of both approaches as the number of tasks is scaled up.

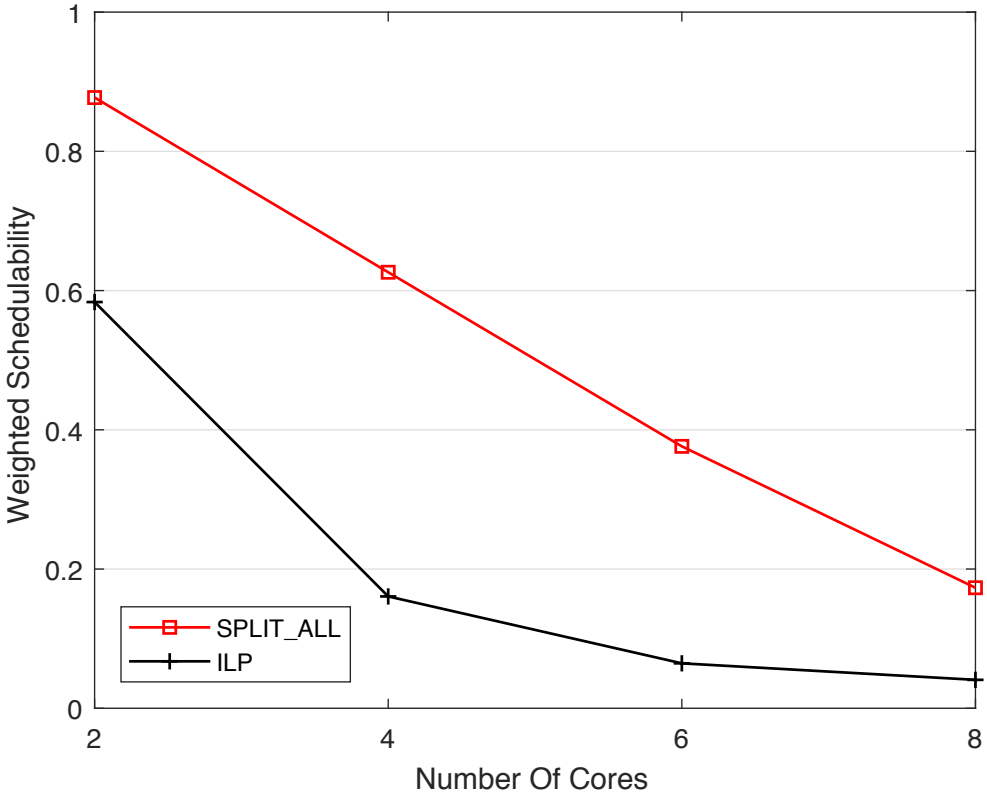


Figure 4.11: A graph utilising weighted schedulability as a metric to determine the performance of both approaches as the number of cores is scaled up.

Experiment Three

Parameters:

- Multiple experimental runs were performed, each increasing the number of cores per set in increments of 2, from 2 to 8.
- Task set size was fixed at 20.
- Tasks were evenly distributed over two criticality levels.
- 4 minor cycles were included per major cycle.

This plot illustrates the impact of scaling the number of cores, the results are shown in Figure 4.11. As we observed in prior experiments, an increase in number of cores, without an increase in number of tasks results in a reduction in WCET. We observe this trend in both the ILP and splitting based approach with the splitting based approach performing better as expected.

Experiment Four

Parameters:

- Multiple experimental runs were performed, each increasing the number of criticality levels by 1, from 2 to 5.
- Allocation was made to a 2 core platform.
- Task set size was fixed at 20.
- 4 minor cycles were included per major cycle.

We illustrate the impact of scaling the number of criticality levels from 2 to 5. The results are shown in Figure 4.12. We observe a slight loss in schedulability as the number of criticality levels is increased, this is due to a new synchronised criticality switching point being added with each additional criticality level. While the splitting approach performs consistently better than ILP, the trend we have observed through multiple experiments of decreasing schedulability as the number of criticality levels is increased holds true.

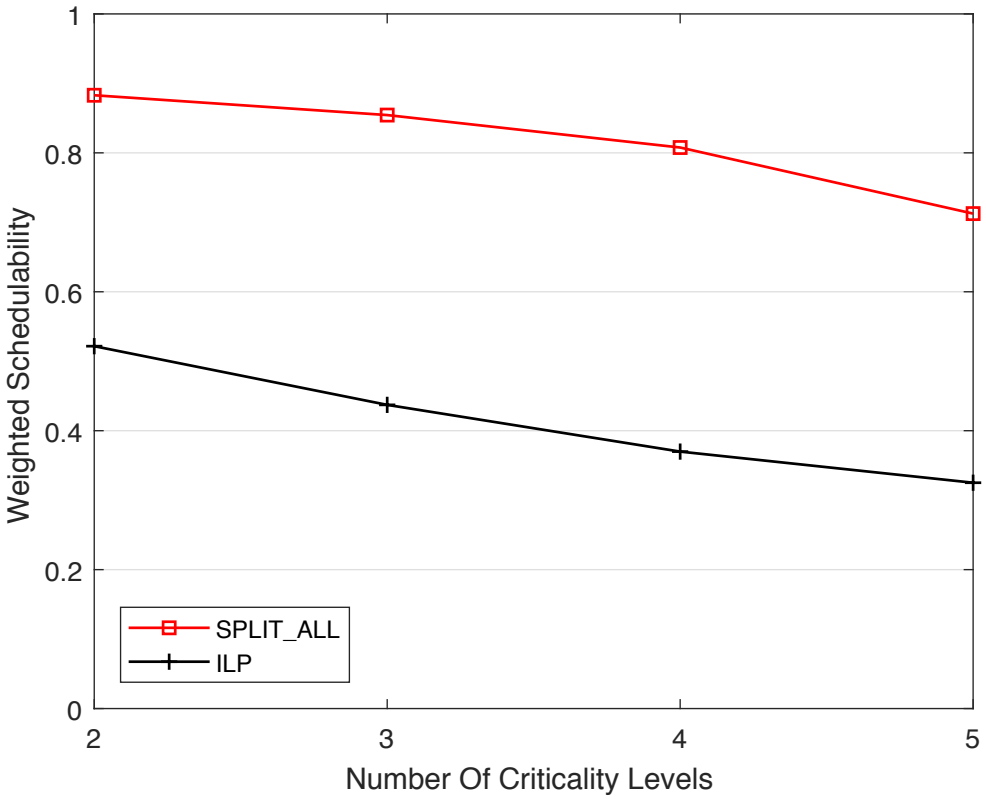


Figure 4.12: A graph utilising weighted schedulability as a metric to determine the performance of both approaches as the number of criticality levels is scaled up.

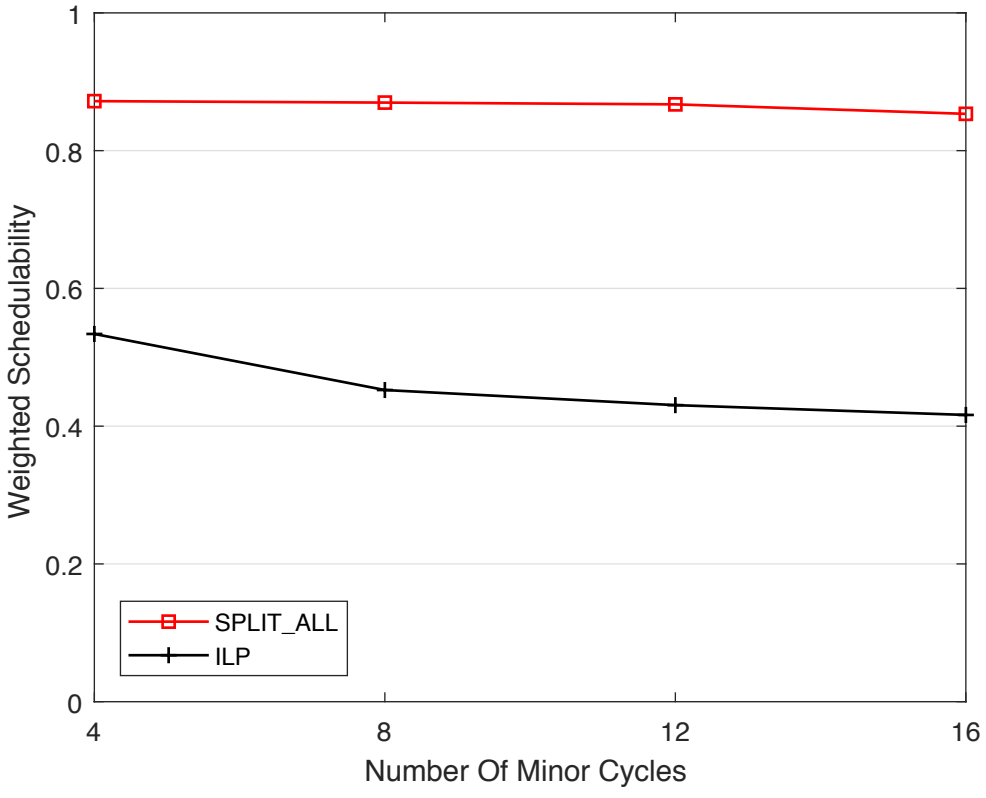


Figure 4.13: A graph utilising weighted schedulability as a metric to determine the performance of both approaches as the number of minor cycles is scaled up.

Experiment Five

Parameters:

- Multiple experimental runs were performed, each increasing the number minor cycles by 4, from 4 to 16.
- Allocation was made to a 2 core platform.
- Task set size was fixed at 20.
- Tasks were evenly distributed over criticality levels.

Finally, this plot illustrates the impact of scaling the number of minor cycles from 4 to 16 per major cycle The results are shown in Figure 4.13. We observe little change in schedulability as the number of minor cycles is increased. This is likely to be due to no change in the periods of tasks, thus if schedulable over 4 minor cycles, the same schedule is copied to the other sets of 4.

Timing

Finally, we report the timing data taken for the Split All and ILP approaches. As we established in Chapter 3, a number of outliers are likely to appear with significantly larger execution times. We report the average, median and maximum recorded execution times for the SplitAll and ILP approaches in Table 4.3. We placed a 60 second limit on the maximum execution time to allow our experiments to complete within a reasonable time.

	ILP	SplitAll
Average	0.0089	1.0206
Median	0.002	0.0098
Max	41.613	60.223

Table 4.3: The average, median and max execution times of the Limited and Unlimited ILP and SplitAll allocation approaches(in seconds).

We see that the median execution time remains low across all approaches. We recorded only 1.5% of task sets having an execution time of greater than 60 seconds. It is clear that these larger outliers exist, however they are the minority of cases. These long execution times prove more of a challenge for experimental work generating large numbers of task sets, rather than a practical implementation

which may focus on only one or two task sets.

In addition to the data in table 4.3, we explored the effect of scaling task set parameters has on execution times. The results of this are reported in Figures 4.14,4.15, 4.16 and 4.17.

Two main points can be drawn from this data:

1. The scaled up approaches tend to require more execution time to find a solution. This is a logical result, as with scaled parameters comes a more complex allocation and models with an increased number of variables.

2. The split approaches tend to require more execution time than the standard ILP. This is due to there being a greater degree of freedom for the solver to find a solution. In addition, to maintain the property of only splitting tasks when required, for HI criticality tasks a simplistic optimisation is used.

Overall we still observe very low execution times per task set across all allocation techniques scaled to all values.

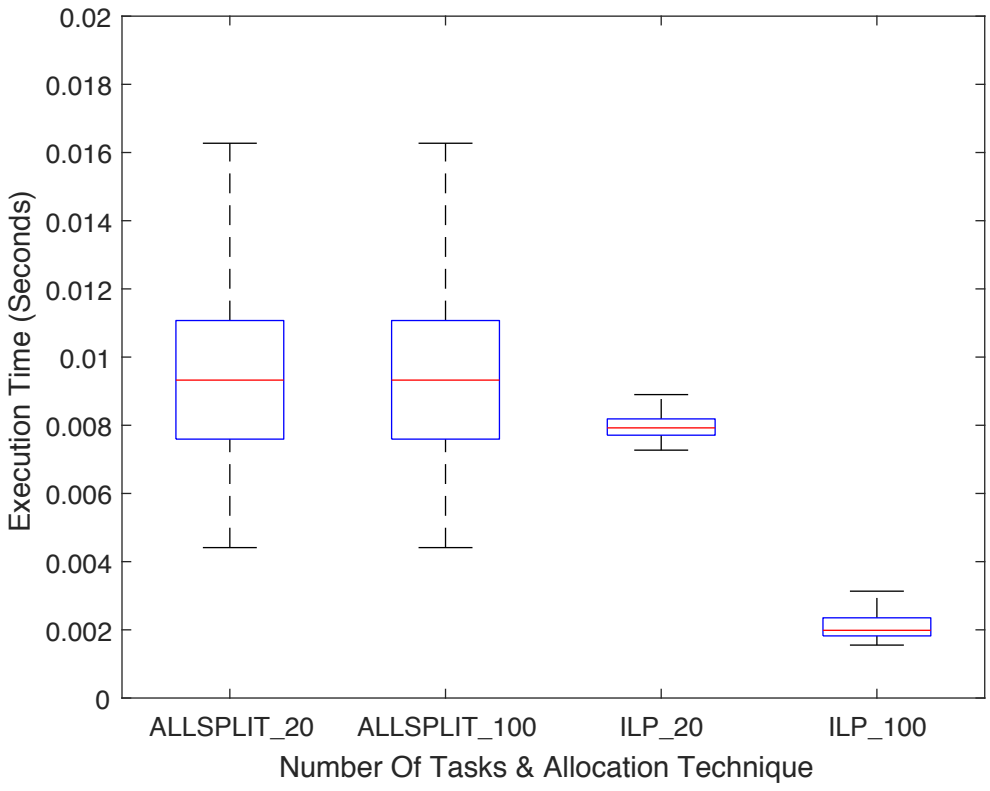


Figure 4.14: A box plot illustrating the execution time as the number of tasks are scaled.

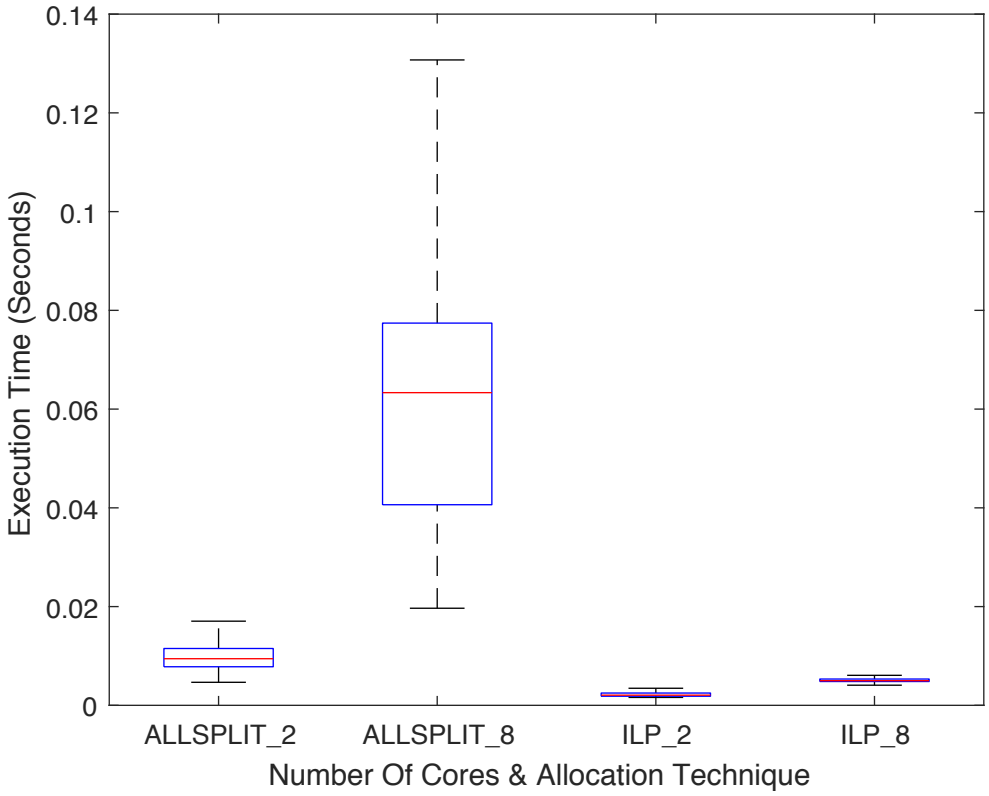


Figure 4.15: A box plot illustrating the execution time as the number of cores are scaled.

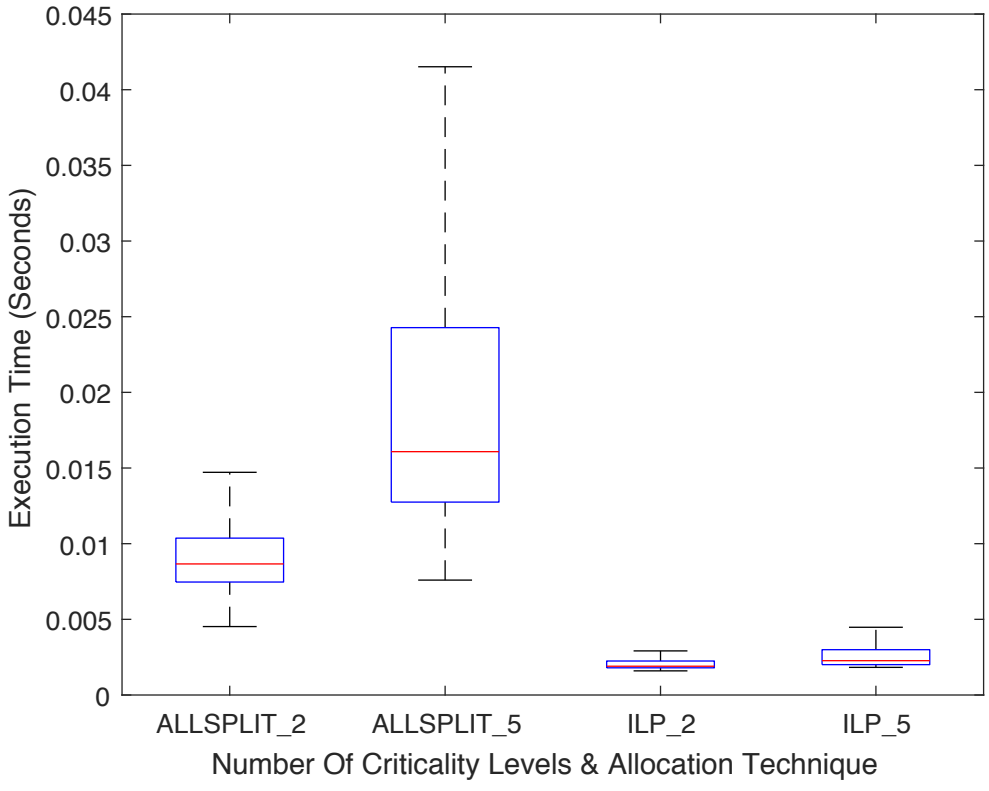


Figure 4.16: A box plot illustrating the execution time as the number of criticality levels are scaled.

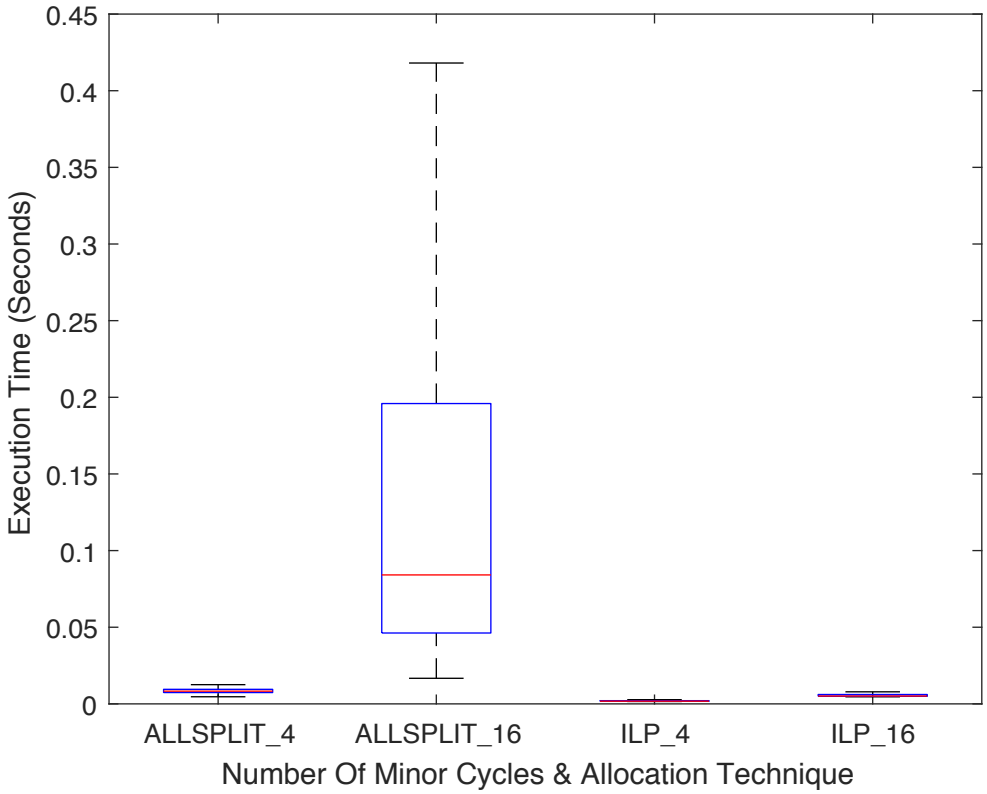


Figure 4.17: A box plot illustrating the execution time as the number of minor cycles are scaled.

Summary

We may draw three key conclusions from these experiments:

- Firstly, It is clear that task splitting does lead to significant increases in schedulability over the base ILP allocation. We illustrated that this improvement holds true when scaled by a number of factors.

- Secondly, we have illustrated that even allowing only a single task to split, can have a significant impact on schedulability. We show that splitting just one task with a high WCET can free up a schedule providing a significant boost in overall schedulability. In fact it seems that the largest impact observed during our experiments was found by splitting just one task with diminishing returns on schedulability for each additional task split. In Figure 4.8, from utilisation intervals 1.3 to 2, splitting a single LO criticality task resulted in an average increase in schedulability of 0.14 and splitting a single HI criticality task resulted in an average increase of 0.15.

- Finally, we illustrated that while the splitting approach does take longer to execute when compared with standard ILP, both approaches complete relatively quickly. Allocations such as this are intended as a system design tool to be utilised during development (offline). Given this use case our techniques complete within a very reasonable time frame (typically less than 0.45 seconds).

In all, we have illustrated that not only is splitting effective, but the selected splitting of just a few tasks can have a significant impact.

4.4 Summary

In summary, this chapter has presented an approach to allow limited task splitting in mixed criticality cyclic executives. An overview of the results of this chapter is provided in the following points:

- We have illustrated how LO criticality splitting is possible by simply allowing the variables defining LO criticality task locations to be continuous rather than integer. The constraints on these variables remain the same, requiring a whole task to be scheduled somewhere across all the variables but allowing the solver to decide on how it is split. We limit splitting to be across minor cycles only.

- Furthermore, we developed a container based approach for splitting HI criticality tasks. We re-envision a HI criticality task as having LO and EX execution times (where EX is the execution time required to go from LO to HI). A container is defined for each and these containers are permitted to split. In addition work allocated to the EX (HI criticality mode) may be allocated to the LO container, the solver decides when this is done.

- Finally, we present some experimental results illustrating the improved schedulability possible with splitting and the notion that only a small number of tasks need be split to gain a significant amount of additional utilisation.

An emphasis should be placed upon the 'limited' aspect of the splitting proposed in this chapter. In some way, this ability to split tasks as described is a schedule

design tool, allowing a system designer to ask 'what if this task was split?'. As illustrated in the experimental results, typically splitting only one or two tasks can lead to large gains in overall system utilisation.

Chapter 5

Optimisations

To this point, we have established a flexible mixed criticality cyclic executive model allocated using forms of Linear Programming (ILP/MLP). However, we have yet to consider how these models may be used to optimise features of the system. This chapter considers two possible optimisations: the first to reduce the number of cores utilised by HI criticality tasks and the second to maximise the spare capacity within a given criticality level.

5.1 Reducing the number of HI criticality cores

Typically, real-time applications of a high criticality level have remained in the single processor domain. Many of these applications are highly sequential and do not extend easily to multi-core platforms. The drive toward integrated mixed criticality architectures may require these, now legacy, sequential applications to be scheduled on modern multi-core platforms. In contrast, newer applications often seek more complex functionality due to the additional resources provided by the new hardware. Such functionality may be highly parallelised, for example image recognition may be a requirement for a number of applications, from targeting assistance in a UAV, to collision avoidance systems in autonomous or semi-autonomous cars.

Broadly speaking we may consider two criticality levels in this scenario. On the one hand, we have HI criticality (often hard real-time) legacy or highly sequential applications. Meanwhile, the platform must also support the LO criticality comparatively complex applications which make use of multi-core architectures. The se-

quential and to some extent, simplistic, nature of the HI criticality work is a product of the verification required to ensure its safety. For those applications requiring very stringent analysis to guarantee safety, their design remains simplistic and sequential, focusing on predictable execution on one core. Scheduling these applications on multi-core platforms is problematic due to uncertain behaviour, as such currently a large degree of pessimism is introduced. The extreme case might provide such pessimistic predictions for a two core platform that only one core must be used. The LO criticality applications are not faced by these restrictions and may often be designed for execution on multi-core hardware.

This section makes the assumption that by reducing the number of cores HI criticality work must execute on, we may reduce the pessimism added to their WCET estimates. The premise is simple, assuming the same cyclic executive platform utilising the barrier protocol for isolation, HI criticality work is executed first on as few cores as possible, (assuming no criticality change) once completed LO criticality work executes on all available cores in the system. As a result of executing HI criticality work on fewer cores we not only reduce the WCET estimates but by proxy reduce the complexity and cost of the analysis.

The proposed model does not represent what is perhaps the long term goal of multi-core real-time platforms, full integration and use of parallelism regardless of criticality level. However, it does propose a mid-ground for near future applications where redevelopment of legacy applications is undesirable, but modern parallelised functionality is required. In addition, the above assumptions are in-line with the thinking of our industrial partners. Crucially multi-core architectures are unavoidable, thus schemes such as this provide a middle way between switching off all but one core, and running all applications in a highly parallel manner.

5.1.1 Quick clarifying examples

Example 1

For clarity the proposed scheme is outlined in this section using an abstract example. Consider the task set shown in Table 5.1.

It contains a mixture of HI and LO criticality tasks to be scheduled on a 3 core

τ	$C(LO)$	$C(HI)$	T	L_i
τ_1	5	10	25	HI
τ_2	5	10	25	HI
τ_3	5	10	25	HI
τ_4	10	15	50	HI
τ_5	15	20	100	HI
τ_6	5	-	25	LO
τ_7	5	-	25	LO
τ_8	5	-	25	LO
τ_9	10	-	50	LO
τ_{10}	10	-	100	LO

Table 5.1: Optimisation: Example 1.

platform with 4 minor cycles (where $T^F = 25$ & $T^M = 100$). The initial schedule (with no optimisation) produced by our standard ILP model is shown in Figure 5.1a.

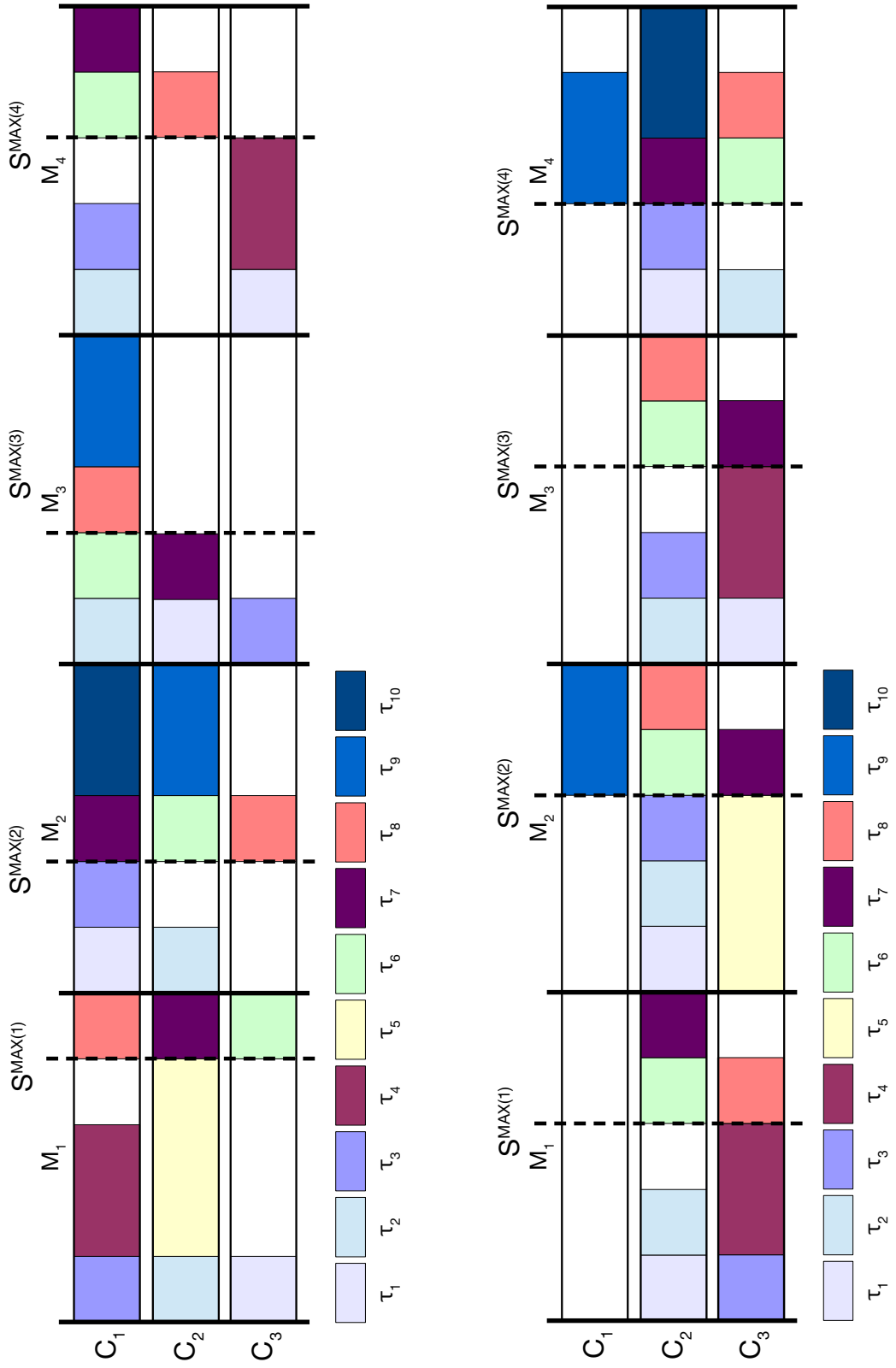
In this figure we can see a mixture of HI and LO criticality work being scheduled across all minor cycles and cores. It is also clear that there is a significant amount of spare capacity (white space), this optimisation aims to use this capacity to tighten the HI criticality schedule. We optimise the model to reduce the number of cores utilised by HI criticality tasks, the resulting schedule is illustrated in Figure 5.1b.

The solver has been able to remove all HI criticality execution from core 1, this has been achieved by relocating much of the slack and, in some minor cycles, postponing the predicted barrier invocation. While we use a 3 core platform in this example, we envision the use of this technique on platforms with a significantly larger number of cores. In such a scenario while the criticality switching point is delayed, the LO criticality tasks would have access to many cores. This provides a platform which might potentially facilitate parallelised applications alongside HI criticality sequential work. This example serves as an illustration of one of the two aspects of this optimisation. It shows how the number of cores for HI work can be reduced, but does not illustrate the effect of the reduction on HI criticality task's WCETs.

Example 2

We now illustrate how the reduction in WCETs for HI criticality tasks when the number of cores used to execute is reduced, can impact the schedulability of a set of tasks on a particular number of cores. Consider the task set in Table 5.2 which

contains two additional HI criticality tasks, τ_{11} and τ_{12}.



(a) A standard (non-optimised) schedule of the task set in Table 5.1.

(b) An allocation, optimised to minimise the number of cores for HI criticality tasks.

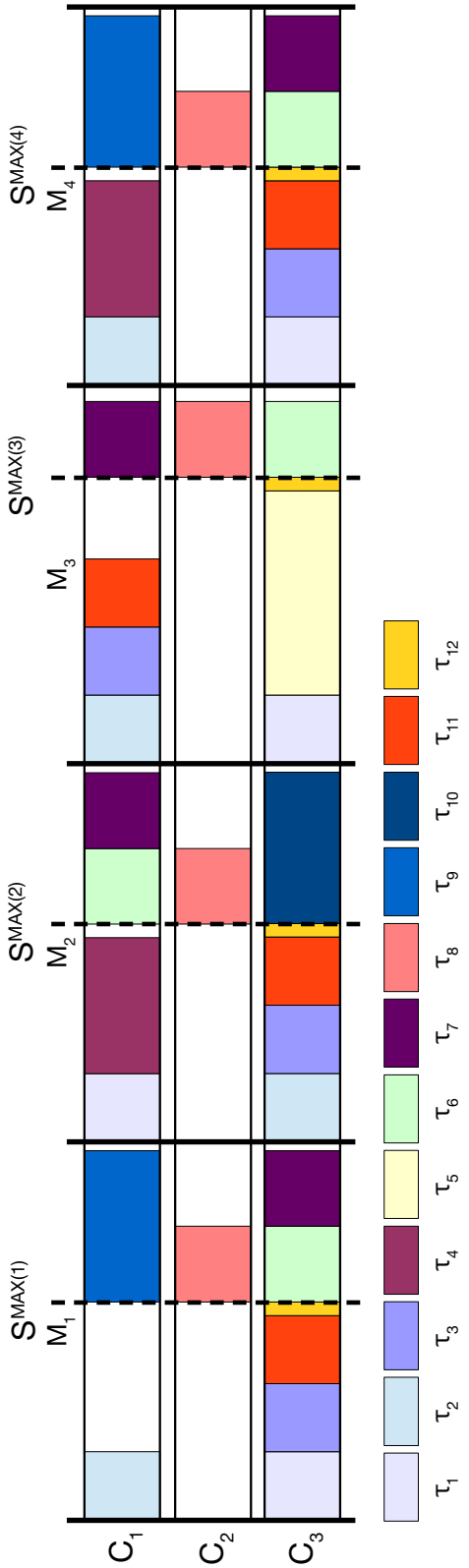
Figure 5.1: 3 Core example schedules.

τ	$C(LO)$	$C(HI)$	T	L_i
τ_1	5	10	25	HI
τ_2	5	10	25	HI
τ_3	5	10	25	HI
τ_4	10	15	50	HI
τ_5	15	20	100	HI
τ_6	5	-	25	LO
τ_7	5	-	25	LO
τ_8	5	-	25	LO
τ_9	10	-	50	LO
τ_{10}	10	-	100	LO
τ_{11}	5	10	25	HI
τ_{12}	2	3	25	HI

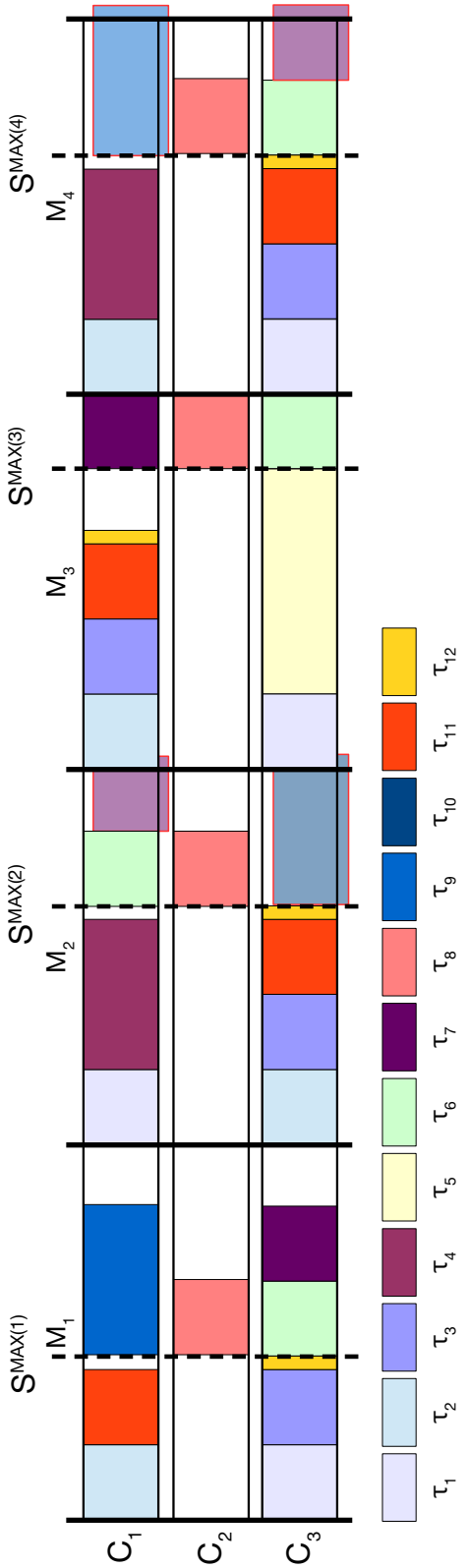
Table 5.2: Optimisation: Example 2.

Given a 3 core platform, this task set may be scheduled such that only 2 cores are used for HI criticality tasks. As the number of cores used by HI criticality tasks has been reduced by 1, we reduce all HI WCETs by 10%. The resulting schedule is shown in Figure 5.2a. This is a compact schedule where only τ_8 executes on core 2, all other tasks are scheduled on cores 1 and 3. No optimisation or constraints have been placed on the LO allocation, more LO tasks may have been allocated to core 2, however they were not in the schedule produced.

If the same schedule is considered, but the 10% reduction is not applied, the following partial schedule is produced, see Figure 5.2b. While during minor cycle 1, we may allocate τ_{11} to core 1 to ensure the schedulability of the minor cycle, in all other minor cycles there is no feasible schedule which limits HI criticality work to 2 cores. In minor cycle 2, τ_{12} prevents τ_7 and τ_{10} from fitting. In minor cycle 3 τ_{12} or τ_1 may be re-allocated to core 1 to make the cycle schedulable. However, minor cycle 4 much like 2, is not able to schedule all HI criticality work on only 2 cores and provide the capacity for all LO criticality tasks. To ensure no other valid schedule existed where a reduction was not made, but schedulability was possible with HI work on 2 cores, we produced the same task set with all WCET values increased by 10%. If a 2 core allocation of this new task set were possible, all WCETs would be reduced by 10%, thus using the real WCETs. Therefore the task set is not schedulable with all HI work on 2 cores at its original WCET values, they must be reduced by 10% in order to fit all HI tasks within 2 cores.



(a) An optimised schedule of the task set in Table 5.2 with 10% WCET reduction per core.



(b) An optimised schedule of the task set in Table 5.2 without the 10% WCET reduction per core.

Figure 5.2: Example schedules illustrating WCET reduction.

5.1.2 Optimisation via ILP

In order to investigate the application of this optimisation we implemented it within our ILP model. This section will detail the additional constraints required. Generally, the optimisation may be split into two components, the first is to minimise the number of cores used by HI criticality tasks, and the second is to model a reduction in verification difficulty by a reduction in HI criticality task WCETs.

Initially, the first requirement was considered. To begin with we defined a binary variable P_z where z is the core. When P_z is set to 1, there is no HI criticality work on that core (during any minor cycle), when it is set to 0 there is HI criticality work. Naturally, additional constraints are required to support this, however, our optimisation becomes simply to maximise $\sum_{\forall z} P_z$.

The key constraint required to implement the P variables will be explained below. For two HI criticality tasks τ_i & τ_l on a 2 core platform with 4 minor cycles the constraints would be:

$$T_i\_11 + T_i\_12 + T_i\_13 + T_i\_14 +$$
$$T_l\_11 + T_l\_12 + T_l\_13 + T_l\_14 + 8 \times P_1 \leq 8$$
$$T_i\_21 + T_i\_22 + T_i\_23 + T_i\_24 +$$
$$T_l\_21 + T_l\_22 + T_l\_23 + T_l\_24 + 8 \times P_2 \leq 8$$

The value 8 is used to both multiply the P variables and constrain the inequalities as 8 is equal to the number of variables other than P in the constraint. It is possible that all variables will be set to 1 (a task with a period equal to the minor cycle), this must be permitted. Crucially if a P variable is to be set to 1, this indicate that no HI criticality work is executing on that core, then it must be multiplied by 8, leaving no space in the inequality for any of the binary location variables to be set to 1. Conversely, if any variable is set to 1 (indicating a task is scheduled on that core) there will be no room for the P variable to be set to 1.

The constraints above represent the basic way, and indeed the first method we implemented, to minimise the number of HI criticality cores. However, it does not support a method for modelling a reduction in WCET. This proved more challeng-

ing requiring significant changes to the model. We detail this technique and its constraints below.

In our model, reducing the number of cores HI criticality work is executing on results in a fixed reduction in all HI criticality WCETs. In this work a value of 10% was chosen simply as an example. We do not claim insight into what the actual value may be and it is likely to be system/platform dependent. To model this we altered the binary location variables making them integers. These integers must be greater than or equal to 0 and less than or equal to 10, for example $0 \leq T_i\_11 \leq 10$. As with the binary variables, setting the variable to equal 0 still indicates that the task is not scheduled in that location, any value from 1 to 10 indicates the task is scheduled. A value of 10 indicates that task is scheduled in that location and that all cores in the system are utilised by HI criticality work, a value of 9 indicates that one core is not used by any HI criticality tasks.

As our variables are between 0 and 10, we must adjust the WCET constraints to account for the correct amount of execution. As such in these constraints we divide the WCET for each HI criticality task by 10 (WCET / 10). Thus when multiplied by a location variable, if that variable is equal to 10 (meaning all cores are executing HI criticality work) then the full WCET will be included. However, if one core is executing no HI criticality work, then the WCET/10 will be multiplied by 9, providing a 10% reduction in WCET ($9 \times WCET/10$). Prior to the variable change the WCET constraint for T_i_11 was:

$$C_i(HI) \times T_i\_11$$

It now becomes:

$$\frac{C_i(HI)}{10} \times T_i\_11$$

To ensure the correct value is included in the location variable depending on the number of cores used by HI criticality tasks we introduce a new set of constraints. Given two variables T_i_11 & T_i_21, if a task must be scheduled in one variable, the constraint would read as follows:

$$T_i\_11 + T_i\_21 + P_1 + P_2 = 10$$

We again assume, for illustration, a dual core setup and thus two processor variables P_1 & P_2 are included. The sum of these variables must equal 10, as the P variables are only set to 1 if no HI criticality work is present on that processor, the value of the location variable will only be reduced from 10 if a core has no HI criticality execution. For example if T_i_11 contained the task, and core 1 had no HI criticality execution, $P\_1 = 1$ thus $T_i\_11 = 9$, therefore the resulting WCET of the task will be reduced by 10%.

To complete the picture we must also update the constraints defined previously to set the P_z variables according to the HI criticality work scheduled on their core. Given two HI criticality tasks τ_i & τ_l we have the following constraints:

$$T_i\_11 + T_i\_12 + T_i\_13 + T_i\_14+$$
$$T_l\_11 + T_l\_12 + T_l\_13 + T_l\_14 + 80 \times P_1 \leq 80$$
$$T_i\_21 + T_i\_22 + T_i\_23 + T_i\_24+$$
$$T_l\_21 + T_l\_22 + T_l\_23 + T_l\_24 + 80 \times P_2 \leq 80$$

The difference is that the value 80 is used rather than 8, 80 is now calculated by the sum of all variables in the constraint, excluding P_z, multiplied by 10. We multiply by 10 to account for our new integer location variables. The constraints function in the same manner, in the extreme case work may be allocated to all location variables on a particular core and still satisfy the constraint. However, the P variables may only be set equal to 1 if no HI criticality work is scheduled on that core.

Constraint updates

Periodicity Constraints

As we now make use of integer location variables our periodicity constraints

must be adapted. Making use of a familiar trick, we require a task + a binary variable $\times 10$ to be less than or equal to 10. We then require a number of those binary variables to equal 1 depending on the period of the task, we use B for these variables. Given two location variables:

T_i_11

T_i_21

Two constraints are defined:

$T_i\_11 + 10 \times B_1 \leq 10$

$T_i\_21 + 10 \times B_2 \leq 10$

As this task's period is equal to the minor cycle, we require the task to be scheduled in one of the two locations (core 1 or core 2 during minor cycle 1):

$B_1 + B_2 = 1$

As we require one of the B variables to equal 1, only one of the location variables may contain a task. Later constraints ensure that the variable is set to an appropriate value (indicated as BVarConstraints in Figure 5.3). A complete set of constraints for τ_i where $T_i = T^F$ is shown:

$T_i\_11 + 10 \times B_1 \leq 10$

...

$T_i\_24 + 10 \times B_8 \leq 10$

$B_1 + B_2 = 1$

$B_3 + B_4 = 1$

$B_5 + B_6 = 1$

$B_7 + B_8 = 1$

WCET Constraints

As mentioned the WCET constraints must be updated to multiply the location variables by the $WCET/10$, this allows the model to make 10% reductions in WCET for each core with no HI criticality execution. This update is required for Stages 1 and 2 (not stage 3 as it concerns only LO criticality tasks).

For Stage 1 (HI criticality WCETs):

$$\frac{C_i(HI)}{10} \times T_i\_11 + \frac{C_l(HI)}{10} \times T_l\_11 \leq T^F$$
$$\frac{C_i(HI)}{10} \times T_i\_21 + \frac{C_l(HI)}{10} \times T_l\_21 \leq T^F$$

And for stage 2 (LO WCETs of HI criticality tasks):

$$\frac{C_i(LO)}{10} \times T_i\_11 + \frac{C_l(LO)}{10} \times T_l\_11 + X_1 \leq T^F$$
$$\frac{C_i(LO)}{10} \times T_i\_21 + \frac{C_l(LO)}{10} \times T_l\_21 + X_1 \leq T^F$$

Additional Constraints

Finally, we define our additional constraints. The first ensures that the P_z variables are set equal to 1 if no HI criticality execution exists on its corresponding core (shown as PVarConstraints in Figure 5.3). While these constraints are mentioned above they are repeated again for convenience:

$$T_i\_11 + T_i\_12 + T_i\_13 + T_i\_14+$$
$$T_l\_11 + T_l\_12 + T_l\_13 + T_l\_14 + 80 \times P_1 \leq 80$$
$$T_i\_21 + T_i\_22 + T_i\_23 + T_i\_24+$$
$$T_l\_21 + T_l\_22 + T_l\_23 + T_l\_24 + 80 \times P_2 \leq 80$$

The second set of additional constraints provides a means of reducing the HI criticality WCET of a task by 10% for each core not used by HI criticality tasks. We

list these constraints for τ_i firstly where $T_i = T^M$:

$$T_i\_11 + T_i\_21 + T_i\_12 + T_i\_22 +$$
$$T_i\_13 + T_i\_23 + T_i\_14 + T_i\_24 + P_1 + P_2 = 10$$

And where $T_i = T^F$:

$$T_i\_11 + T_i\_21 + P_1 + P_2 = 10$$
$$T_i\_12 + T_i\_22 + P_1 + P_2 = 10$$
$$T_i\_13 + T_i\_23 + P_1 + P_2 = 10$$
$$T_i\_14 + T_i\_24 + P_1 + P_2 = 10$$

Model Updates - Overview

We present an updated model overview in Figure 5.3:

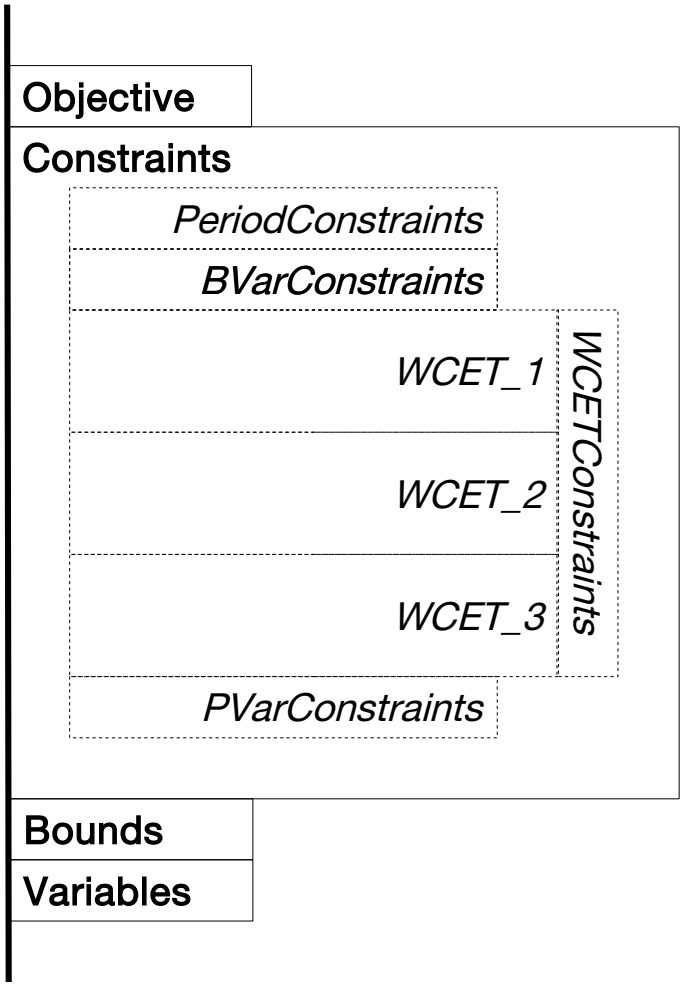


Figure 5.3: Abstract Diagram: Optimisation Constraints.

5.1.3 Experiment: An optimisation to reduce the number of cores utilised by HI criticality tasks

These experiments are designed to investigate the impact this optimisation can have on the number of cores used by HI criticality tasks in comparison to non-optimised allocations. Two experiments are performed, the first seeks to understand how the number of cores used by HI criticality work can be reduced and the second investigates any improvement in schedulability due to reductions in HI criticality task's WCETs when scheduled on fewer cores. In the case of both experiments we consider both a standard and scaled up set of parameters to explore the effect. Finally, as we have moved to performing optimisations we present timing data detailing the run-time of the approach.

Setup

The setup for these experiments was as follows:

- 1000 task sets were generated per 5% utilisation increase.
- Our standard set of tasks included 40 per set, with a scaled up version using 50.
- The standard experiments utilised 4 cores and the scaled up experiments utilised 8.
- We used 2 criticality levels for all experiments.
- Task utilisations were generated using UUniFast, an algorithm presented in [25] which provides an unbiased distribution of utilisation values, following standard practice in synthetic task set generation.
- The minor cycle length was set at 25, with the major cycle length set at 100 ($T^F = 25, T^M = 100$)
- Task periods were selected at random from either 25, 50 or 100.
- Deadlines were set equal to periods. $D_i = T_i$.

- The LO execution times of each task were produced as follows: $C_i(LO) = U_i/T_i$

- For tasks with a criticality greater than the lowest, their HI execution times were determined by $C_i(L_i) = C_i(LO) * CF$ - CF is the criticality factor, a random value between 1.2 and 2.

- Timing data was recorded to find the average time each approach took to find a solution.

- The barrier protocol was implemented for all allocation techniques.

- A reduction in WCET, of 10% per core that HI criticality work does not execute on, is included to simulate a decrease in pessimism when fewer cores are used.

Results

These results are split into two experiments. The first illustrates the reduction in the number of cores utilised by HI criticality tasks as a result of this optimisation. The second investigates the schedulability improvement due to reduced WCETs.

Experiment One

This experiment explores the number of cores used by HI criticality tasks as the utilisation of the task sets generated is increased. We compared the optimised and non-optimised models. Our first plot in Figure 5.4 illustrates how optimising to reduce HI criticality cores has significant impact in comparison to the non-optimised models which utilised all cores for HI criticality work right from the lowest utilisation value. Figure 5.5 illustrates the same effect but on a system where the number of tasks was equal to 50 per set and the number of cores equal to 8. This plot shows our optimised model utilising at most 5 cores for HI criticality tasks whereas the non-optimised approach again uses all cores for HI criticality work right from the lowest utilisation.

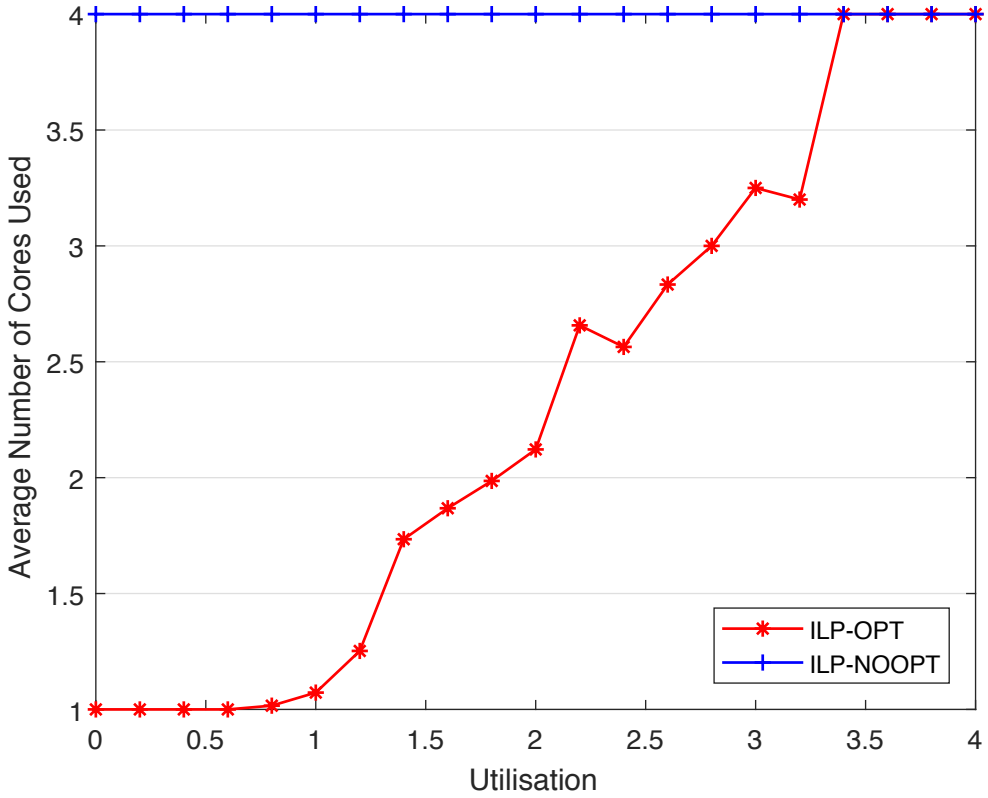


Figure 5.4: A plot showing the average number of cores used by HI criticality tasks at each given utilisation (A lower line = better performance).

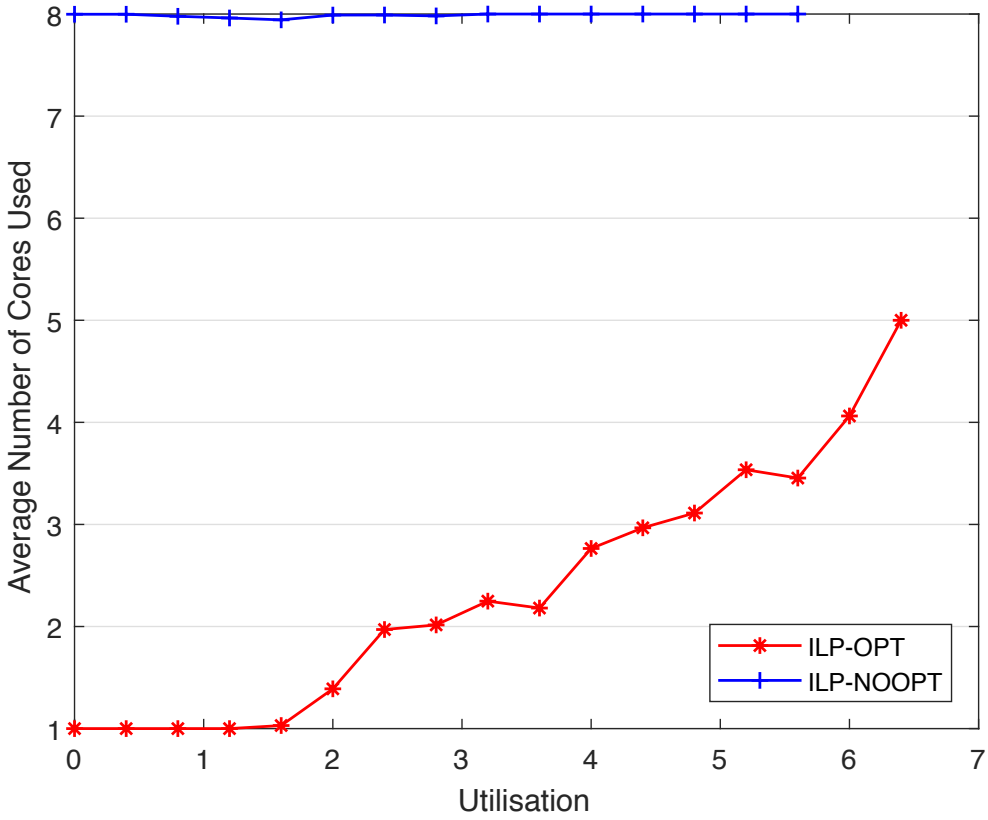


Figure 5.5: A plot showing the average number of cores used by HI criticality tasks at each given utilisation on a scaled up system (A lower line = better performance).

Experiment Two

Experiment two considers the impact on schedulability as a result of the 10% WCET reduction for HI criticality tasks. We first consider the schedulability of our standard configuration with 40 tasks and 4 cores, this is shown in Figure 5.6. We can see here that our optimised approach boasts a slightly improved schedulability over the non-optimised due to the WCET reductions when using fewer cores for HI criticality work. We also investigated the scaled up experiment with 50 tasks and 8 cores, the results are shown in Figure 5.7. Here we see an even greater improvement in schedulability due to WCET reductions, perhaps due to the larger number of cores and the maximum of 5 used for HI criticality tasks as shown in Figure 5.5.

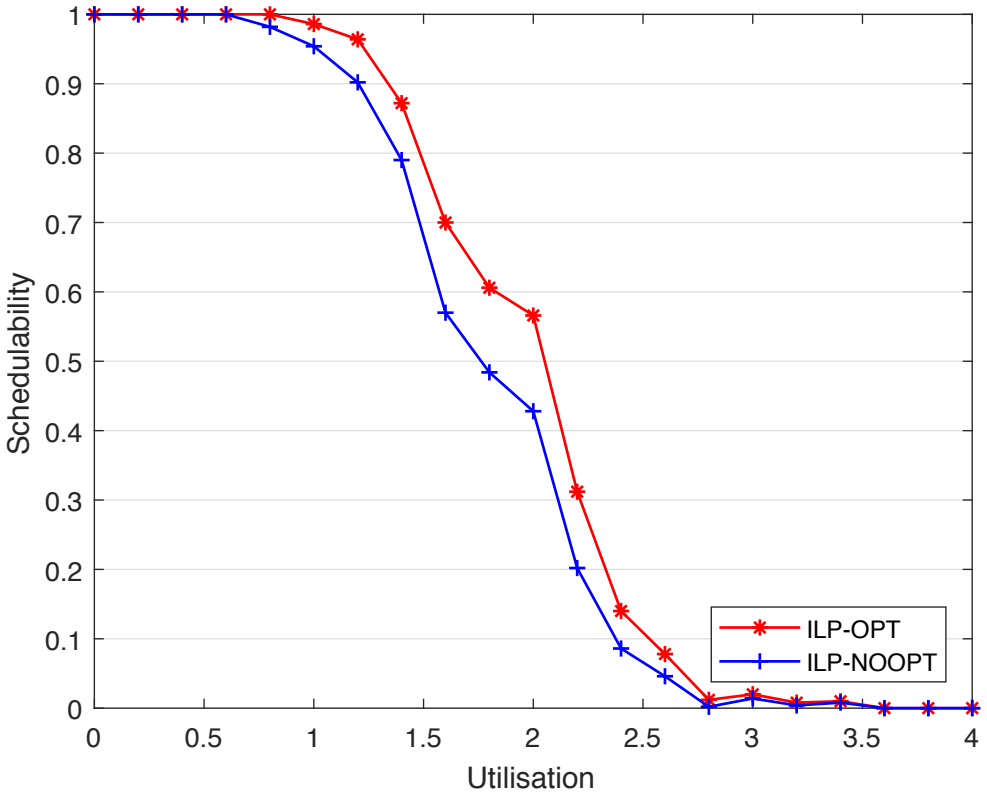


Figure 5.6: A standard schedulability plot illustrating the difference between the optimised and non-optimised ILP models.

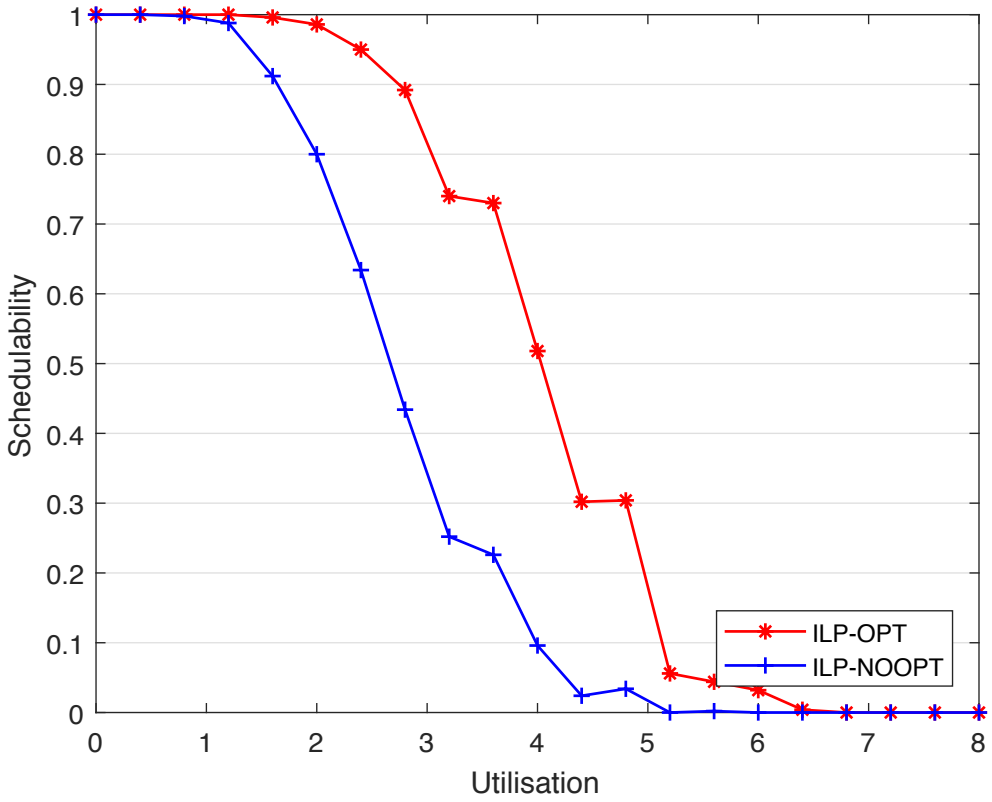


Figure 5.7: A standard schedulability plot illustrating the difference between the optimised and non-optimised ILP models on our scaled up system.

Timing

As we are now performing an optimisation, we re-visit the investigation into the run-time of our LP models. As we have done before, we recoded the average, median and maximum observed execution times. All approaches have an execution time limit of 60 seconds per task set, we observe only 0.19% and 1.33% of the standard and scaled task sets (respectively) reach this limit. We exclude these values from the box plot as they dilute the result. These are presented in Table 5.3:

	ILP-NOOPT	ILP-NOOPT(Scaled)	ILP-OPT	ILP-OPT(Scaled)
Average	0.0026	0.0114	0.1685	1.1606
Median	0.0026	0.0109	0.0067	0.0686
Max	0.0724	0.0744	60.132	60.222

Table 5.3: The average, median and max execution times of the optimised and non optimised allocations (in seconds).

We observe that, while the optimisation does increase the execution time, the average and median times remain very low for both the standard and scaled approach. As mentioned, only a very small percentage of tasks each the execution time limit.

In addition we present a box plot illustrating the difference between the standard and scaled allocation approaches. The box plot in Figure 5.8 shows that while execution times have increase substantially with a good number of models taking over 1 second to solve, overall our ILP models solve within very reasonable time frames.

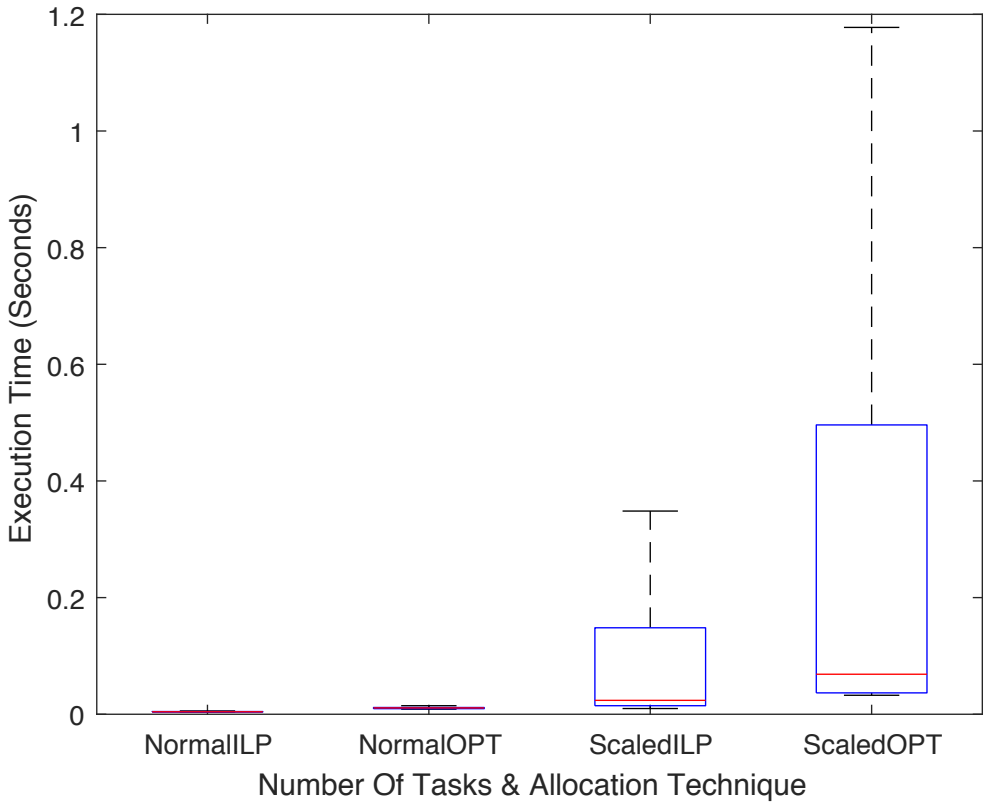


Figure 5.8: A box plot illustrating the timing requirements for this optimisation.

Summary

These experiments illustrate firstly that in a large number of cases, the number of cores use by HI criticality work can be significantly reduced. Secondly we show how the reduced number of cores used by HI work, resulting in reductions in WCET estimates can improve the schedulability of the system over a larger number of task sets.

5.2 Maximising Capacity on Either Side of the Barrier

In this section we consider a simple but potentially valuable optimisation. This optimisation seeks to maximise the capacity on either side of the barrier, LO or HI. While we consider the dual criticality case, this principle is applicable to multiple levels of criticality. We describe the maximisation of the LO and HI capacity separately, following up with some experimentation illustrating the effect of both.

5.2.1 Maximising LO capacity

The aim of this optimisation is to maximise the LO criticality capacity in the system. In other words, in a dual criticality system, we aim to maximise the capacity to the right of the barrier invocation. We illustrate this in Figure 5.9 for clarity, the hatched space is the space we seek to maximise.

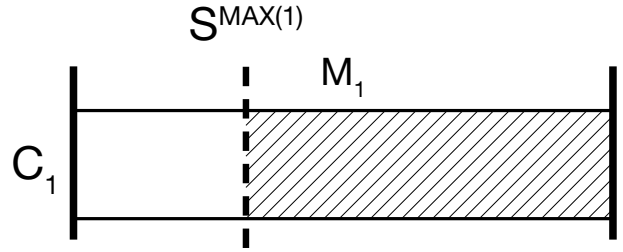


Figure 5.9: A single minor cycle illustrating where the LO capacity is located.

Maximising such capacity has multiple benefits. Firstly, and perhaps most clearly, with additional LO capacity, it is possible and easy to see how additional LO criticality tasks may be scheduled, and what parameters such tasks may take. Secondly, the LO criticality tasks themselves may become more tolerant to a later invocation of the barrier than was pre-computed offline. Even if HI criticality work executes beyond the predicted barrier invocation (thus changing the criticality level of the system) the spare capacity created on the LO side of the barrier may make these tasks more resilient. Finally, as motivated by an industrial example (see Chapter 6) a situation may occur where tasks have potentially unbounded execution times, in this scenario, any capacity moved to the LO criticality mode can be used to enhance their quality of service.

We will explore how the creation of additional LO capacity can aid in the introduction of new LO criticality tasks using the task set shown in Table 5.4:

τ	$C(LO)$	$C(HI)$	T	L_i
τ_1	5	10	25	HI
τ_2	5	10	25	HI
τ_3	5	10	25	HI
τ_4	10	15	50	HI
τ_5	15	20	100	HI
τ_6	5	-	25	LO
τ_7	5	-	25	LO
τ_8	5	-	25	LO
τ_9	10	-	50	LO
τ_{10}	10	-	100	LO

Table 5.4: Optimisation: Max LO Example.

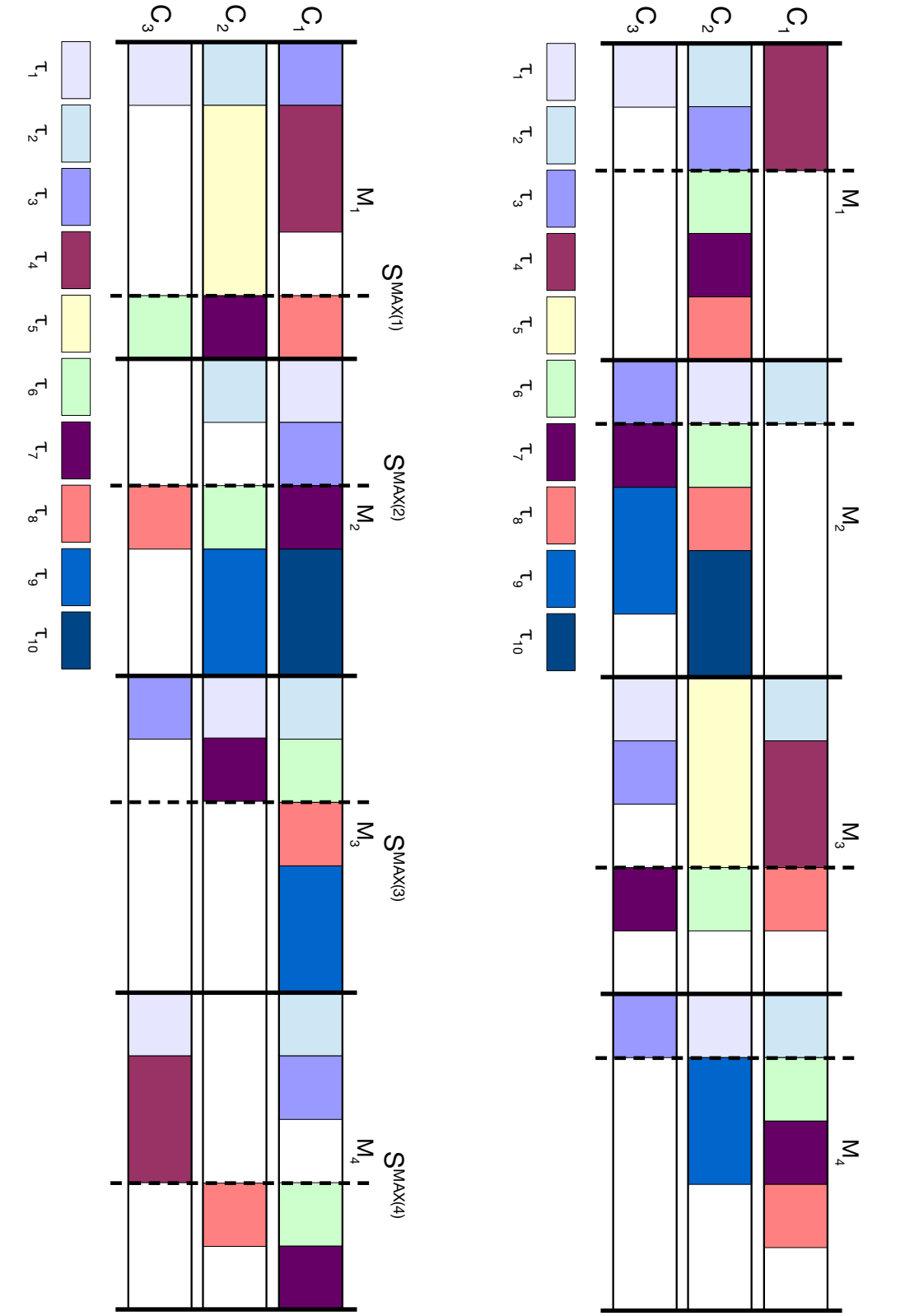
This set of tasks is designed to execute on an 3 core platform and is the same set as seen in Figure 5.1. The schedule generated when the task set is formulated as an ILP model with no optimisation goal is shown in Figure 5.10a.

It is clear from the significant sections of 'white space' both to the left and right of the predicted barrier invocations that there is a large amount of spare capacity. With our optimisation we aim to move this capacity from the left to the right hand side as much as possible. Consider the case where we, as system designers, are tasked by a client to introduce two additional LO criticality tasks with significant computational requirements. We may use such an optimisation to understand the capacity available to us which can help influence the design of the tasks. The schedule where the ILP solver maximises the LO criticality capacity is shown in Figure 5.10b.

Given that we represent the capacity for LO criticality tasks using a variable, X (one for each minor cycle, see Chapter 3), we must simply maximise the sum of the X variables to effectively maximise the LO criticality capacity. A caveat to this is that there is no requirement for any of the spare capacity to be distributed in some fair manner across minor cycles. The solver simply seeks to find the allocation with the maximum spare capacity. The objective function required is listed below for a system with 4 minor cycles:

$Maximise$

$X_1 + X_2 + X_3 + X_4$



(a) A standard (non-optimised) schedule of the task set in Table 5.4.

(b) A schedule from the task set in Table 5.4 optimised to maximise the LO criticality capacity.

Figure 5.10: Schedules investigating LO criticality spare capacity maximisation.

5.2.2 Maximising HI capacity

While maximising the HI criticality capacity is a similar process to that of maximising LO some additional motivation and points may be discussed. Firstly, for clarity, we are interested in maximising the spare capacity to the left of the barrier, see Figure 5.11 for an illustration:

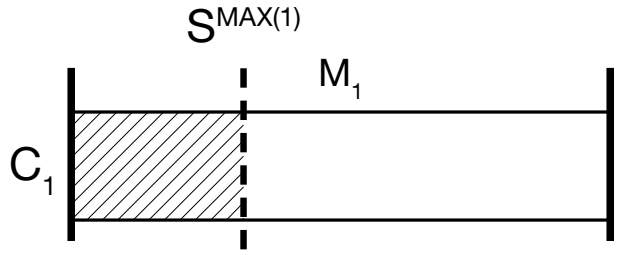


Figure 5.11: A single minor cycle illustrating where the HI capacity is located.

The motivation regarding the addition of new, this time HI criticality tasks, remains the same as discussed for the LO case. However, we also consider the implication of using the spare capacity to offset the statically pre-computed (offline) points at which under normal execution the barrier protocol is expected to trigger. By artificially increasing this value, we reduce the likelihood of the system changing its criticality level due to an overrun of the predicted barrier invocation. Consider Figure 5.12, this shows a schedule from the tasks in Table 5.4, optimised to maximise HI criticality capacity. It is clear that a large amount of the available spare capacity has been moved into the HI criticality mode (indicated by the white space to the left of each barrier).

Implementing this optimisation is a simple case of reversing the ILP objective statement from the LO case. Rather than maximising the X variables, we seek to minimise them, and thus include spare capacity on the HI criticality side of the barrier. The constraints are defined below:

$Minimise$

$X_1 + X_2 + X_3 + X_4$

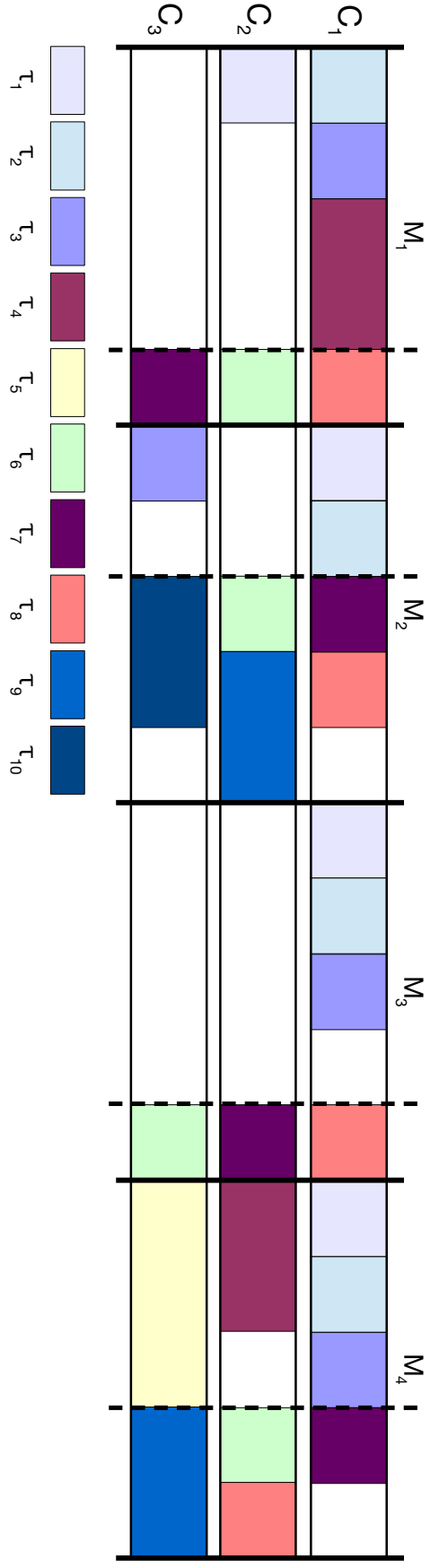


Figure 5.12: A schedule from the task set in Table 5.4 optimised to maximise the HI criticality capacity.

5.2.3 Multiple Criticality Levels

As mention our approach is also applicable to task sets with greater than 2 levels of criticality. Very little changes, the optimisation still seeks to maximise the capacity during a single criticality level. The solver would ensure that the allocation of the other criticality levels (those not being maximised) was as tight as possible. This is the same principle as that seen above (in section 5.2.1), if the LO criticality capacity is maximised, the HI criticality tasks are packed as tightly as possible. Practically this involves maximising (or minimising if the highest criticality level) the appropriate X variables for the criticality level whose capacity is to be maximised.

5.2.4 Experiment: HI & LO Capacity Gains

The purpose of this experiment was to investigate how much spare capacity can be gained by optimising the HI or LO capacity either side of the barrier. We investigate maximising LO capacity and maximising HI capacity separately, a non-optimised model is also measured to understand the gains in spare capacity.

Setup

The setup for these experiments was as follows:

- Our standard set of tasks included 20 per set.
- The standard experiments utilised 2 cores.
- We used 2 criticality levels for all experiments.
- Task utilisations were generated using UUniFast, an algorithm presented in [25] which provides an unbiased distribution of utilisation values, following standard practice in synthetic task set generation.
- The minor cycle length was set at 25, with the major cycle length set at 100 ($T^F = 25, T^M = 100$)
- Task periods were selected at random from either 25, 50 or 100.
- Deadlines were set equal to periods. $D_i = T_i$.

- The LO execution times of each task were produced as follows: $C_i(LO) = U_i/T_i$
- For tasks with a criticality greater than the lowest, their HI execution times were determined by $C_i(L_i) = C_i(LO) * CF$ - CF is the criticality factor, a random value between 1.2 and 2.
- Timing data was recorded to find the average time each approach took to find a solution.
- The barrier protocol was implemented for all allocation techniques.
- Tasks were evenly distributed across criticality levels.

Results

Experiment One

Parameters:

- To produce a suitable example curve 1000 task sets were generated per 5% utilisation increase.

Initially we simply plot the schedulability result of this experiment using a standard schedulability X utilisation graph. We do this to illustrate the performance we expect and to show when the solver fails to find any feasible allocations. The plot is shown in Figure 5.13.

Experiment Two

Parameters:

- A smaller sample of 50 task sets were generated per 5% utilisation increase.

Next we begin by demonstrating the spare LO criticality capacity of the optimised and non-optimised approaches. We plot the average spare capacity at each utilisation point for both the optimised and non-optimised model. The results are shown in Figure 5.14. We can see clearly that as the utilisation is increased, the spare capacity available to the scheduled produced by our optimised model gradually reduces as the system is greater utilised. In addition, we see the non-optimised model following no particular pattern and remaining consistently low.

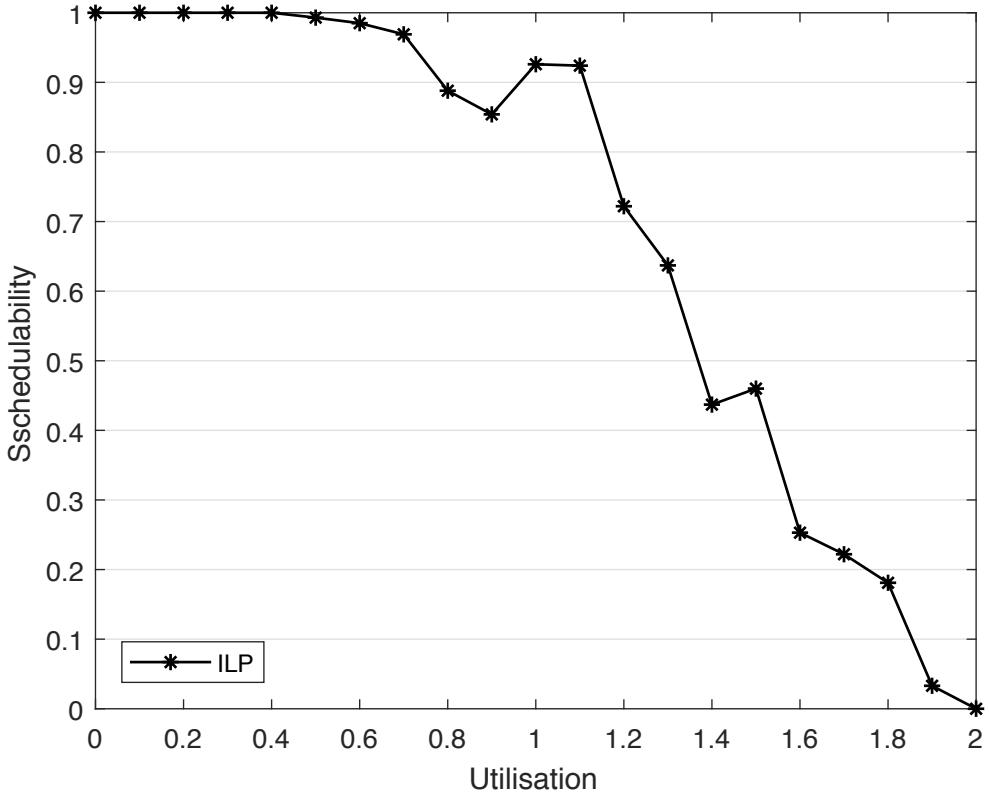


Figure 5.13: A standard schedulability plot illustrating performance.

We performed the same experiment but this time the aim is to maximise the HI criticality capacity. The results in Figure 5.15 differ slightly, while we do observe the same gradual decline in the % of spare HI criticality capacity as we did with the LO, the standard approach actually allocates a good deal of spare capacity to the HI mode. The non-optimised approach does not perform as well as the optimised, however there appears to be some feature of the model which causes the natural allocation to favour increasing HI criticality capacity.

Experiment Three

Parameters:

- A smaller sample of 50 task sets were generated per 5% utilisation increase.
- Tasks were allocated across 2 cores giving a maximum possible spare capacity of 200.

The final experiment seeks to further investigate the performance of our optimisation. To do this we plot the raw data for the LO and HI capacity in the optimised and non-optimised cases. The result for the LO criticality optimisation is shown in Figure 5.16.

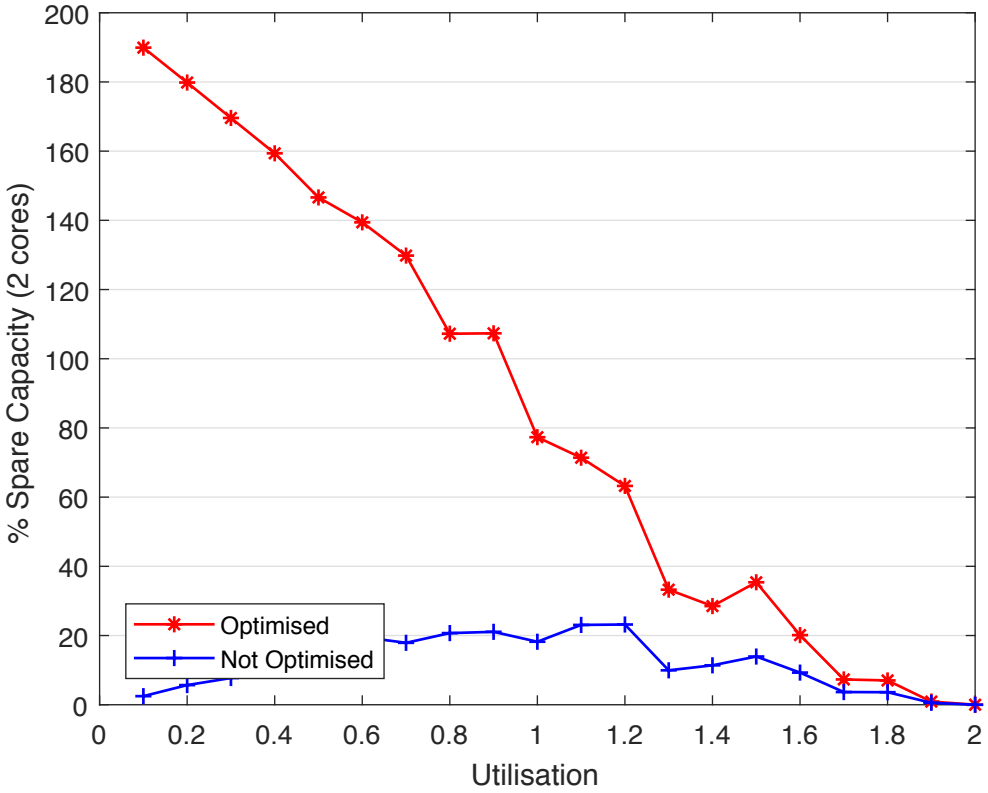


Figure 5.14: A plot showing the spare LO criticality capacity available as the utilisation is increased (higher is better).

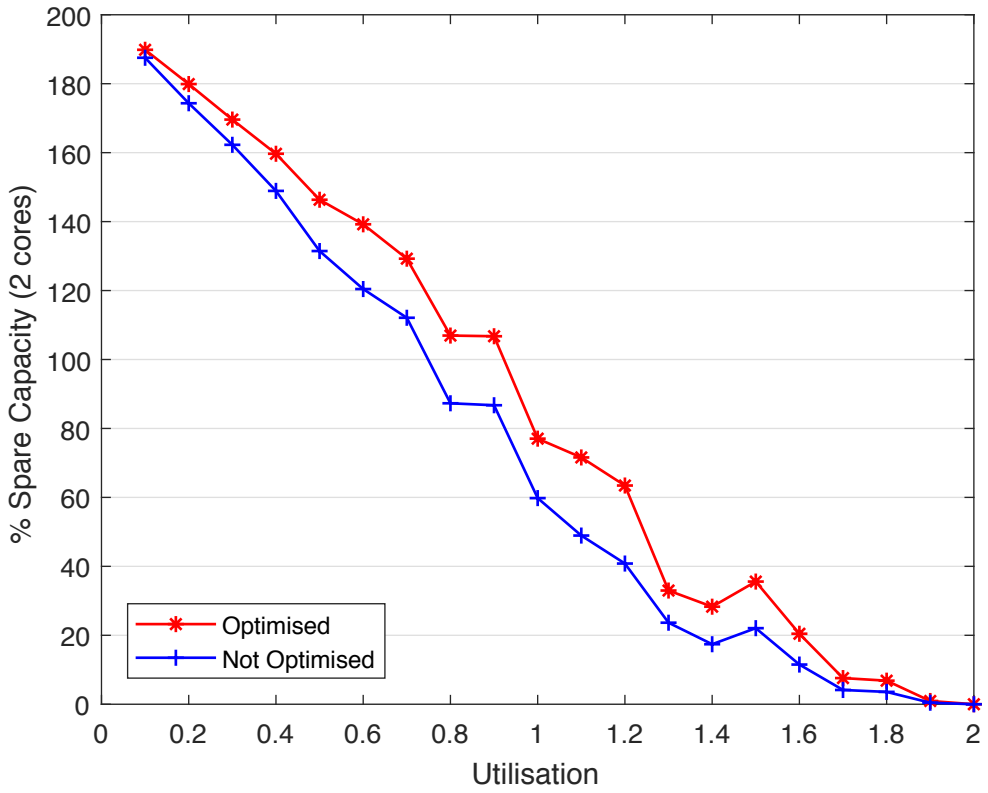


Figure 5.15: A plot showing the spare HI criticality capacity available as the utilisation is increased (higher is better).

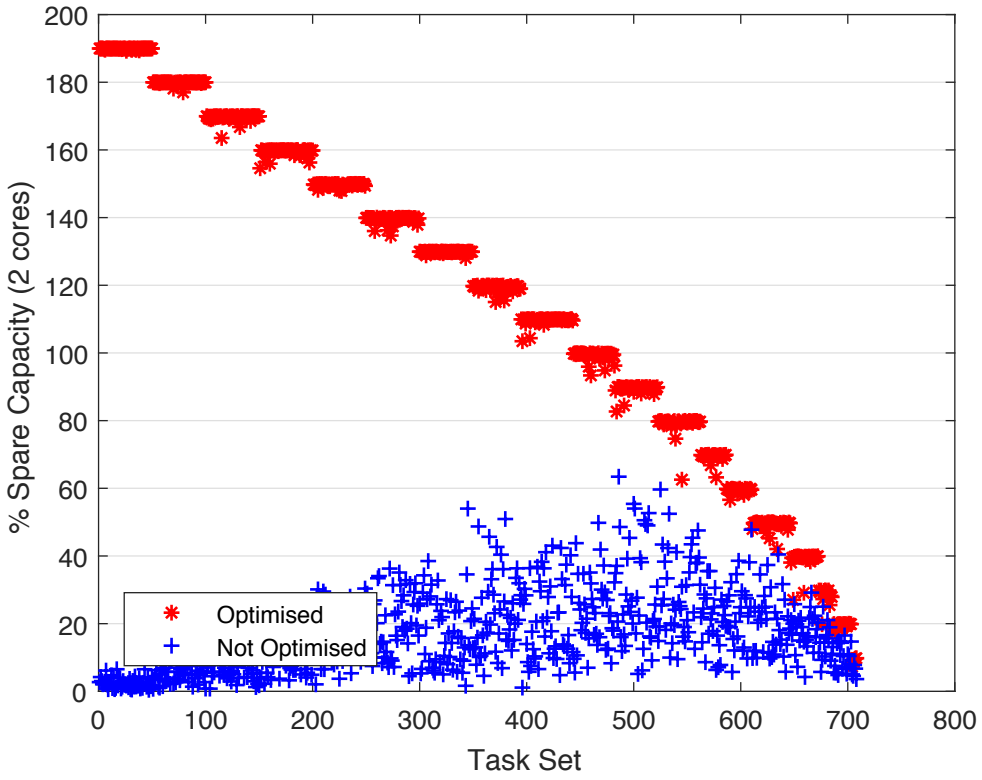


Figure 5.16: A plot illustrating the result of each task-set optimised to maximise LO criticality spare capacity, numbered in the order generated.

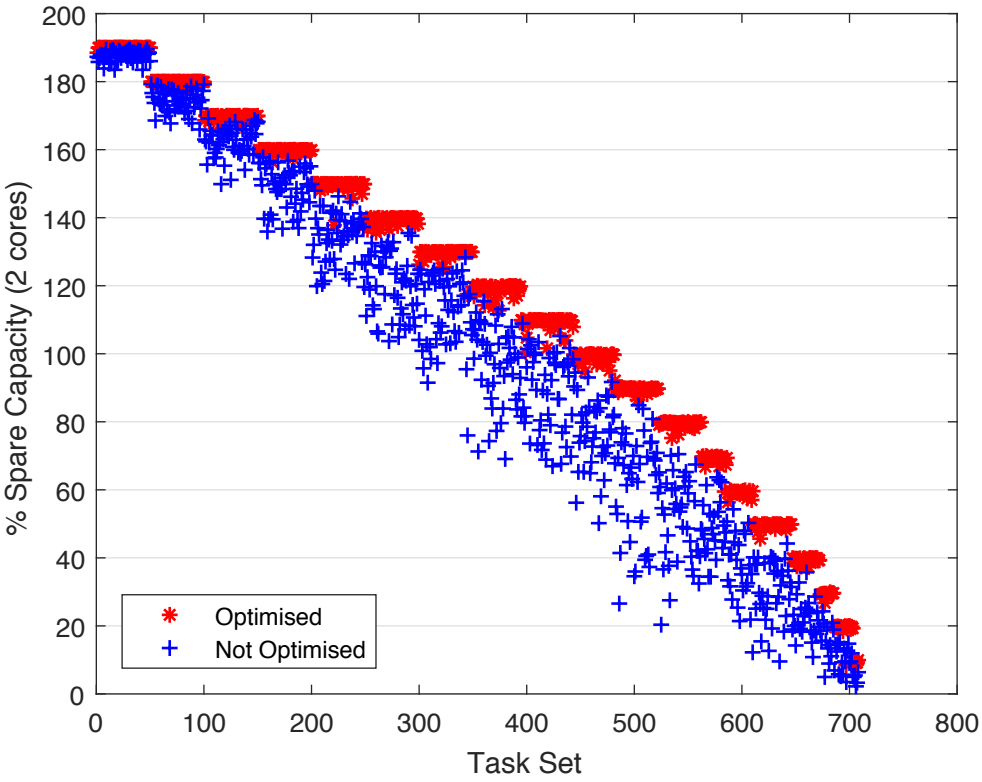


Figure 5.17: A plot illustrating the results of each task-set optimised to maximise HI criticality spare capacity, numbered in the order generated.

We may make two key observations. Firstly, it is clear that there is no particular pattern to the allocation of spare capacity in the non-optimised case. This is to be expected given that it has no optimisation goal. Secondly, the clustered optimised points illustrate that for each of the task sets tested, we optimised most sets to contain a very similar amount of spare LO capacity. Each step down shown by the optimised result illustrates a step up in utilisation. In this plot we can clearly see the optimisation performing well and managing a mush more consistent result across all sets optimised.

We plot the same graph for the optimisation to maximise HI criticality capacity. This is shown in Figure 5.17.

This plot illustrates the same result for the optimised model. We see clear steps down in spare capacity as the utilisation is increased and again we see that almost all of the sets optimised perform close to this maximum spare capacity found at each step. The non-optimised approach differed here slightly, it is clear that there is some tendency to allocate spare capacity to the HI criticality mode, but the results are variable and represent no particular pattern.

Timing

Finally, we observe the execution time required for this optimisation. The results of the timing taken from both optimised approaches and the non-optimised ILP is shown in Table 5.5.

	ILP	ILP-LOOPT	ILP-HIOPT
Average	0.042	5.8	7.909
Median	0.002	0.45	0.533
Max	38.296	451.65	1902

Table 5.5: Timing results for the experiment to maximise LO/HI capacity (in seconds).

The timing results for this optimisation are curious for a number of reasons. Firstly, we observe that the average execution time does increase significantly from what we have seen previously for both optimised approaches. However, we do observe that the median execution time for all approaches is relatively low, drawing

the conclusion that a number of significant outliers must be behind the higher averages seen for the optimised models. Infact, while we did not implement an execution time cutoff, if we assume a value of 60 seconds, only 1.8% of ILP-LOOPT and 1.7% of ILP-HIOPT execution times exceeded 60 seconds. While these outliers are high, overall the approach is able to perform the optimisation quickly on the vast majority of task sets. The removal of the 60 second cap is likely to significantly increase the Max value recorded in Table 5.5. In some way we are observing an expected increase in execution cost as we move from feasibility tests to models with an optimisation function.

Summary

We summarise the conclusions to this experimentation in a number of points.

- Firstly, we have illustrated that our optimised models perform well and are able to effectively maximise the spare LO and HI criticality capacity of a given task set.
- Secondly, we illustrated that this optimisation manages to optimise each model to provide a similar level of spare capacity within each utilisation step.
- Finally, we have illustrated an increase in execution cost for this optimisation. While the increase is significant, it was to be expected as we move away from feasibility tests and into more complex optimisations.

In conclusion we have illustrated how this optimisation can better provide and inform on the spare capacity in a particular criticality mode. We illustrated the improved capacity over standard ILP and explored how the raw results of each of the optimised models cluster to a stepping effect at each utilisation interval.

5.3 A Note on Model Generation

Throughout the prior chapters we have discussed the use of a tool to generate ILP/MLP models and execute them to find feasible task to core/minor cycle allocations. In this section, we discuss how this model generation functions and provide

an overview of what is required. Up to this point, this thesis has described ILP/MLP models using the .lp format supported by Gurobi[45]. However, this format is used purely for illustration, in reality models are created and executed within Matlab [62] using the Gurobi API[45].

Within Matlab we define an object called *Model*, this is used to contain all the information required by the solver to understand and solve the allocation.

Figure 5.18: The empty model object.

Model generation begins with the creation of a sparse matrix, a matrix composed of mostly 0 values. This matrix specifies both the variables and the constraints placed on those variables. Each additional column introduces a new variable while each additional row defines a new constraint. We present a matrix below which defines four variables and two constraints (such a matrix is defined in Matlab and converted to a sparse matrix).

1 1 0 0

0 0 1 1

Given the information in this matrix, we understand the following (presented in the .lp format):

- Our model contains 4 variables, Q1, Q2, Q3, Q4.
- Our model contains 2 constraints

  1. $1 \times Q1 + 1 \times Q2\ (<, >, \leq, \geq, =)\ N$
  2. $1 \times Q3 + 1 \times Q4\ (<, >, \leq, \geq, =)\ N$

  Where N is the value which constrains the variables.

By changing the values from 1 to another number, we alter the value which the variable is multiplied by in that particular constraint. We represent this sparse matrix as part of the model object as $.sparesM$.

Following the definition of our sparse matrix, we provide a vector of values, one for each row in the matrix, which define the value N, on the right-hand side of each inequality. In this case, we define a vector:

1 1

Thus our constraints now read:

$1 \times Q1 + 1 \times Q2(<, >, \leq, \geq, =)1$

$1 \times Q3 + 1 \times Q4(<, >, \leq, \geq, =)1$

This vector of constraint values is recorded in the model as $.rhs$.

To complete the definition of these constraints an additional vector is defined containing $<, >, \leq, \geq or =$. We define such a vector for our model:

> >

This is added to the model as the vector $.sense$.

Next we provide vectors, this time one value for each column (variable) which define upper and lower bounds respectively.

- Upper Bound:

  1 1 1 1

- Lower Bound

  0 0 0 0

We add these vectors to the model object as $.ub$ and $.lb$.

Similarly, a vector is defined with a value for each column, which specifies the variable type. Such types can include I for integer, s for continuous and b for binary. We define our vector:

I I I I

This vector is included in the model object as $.vtype$.

We must define our $.obj$ vector which contains one value for each column. This vector specifies the minimise or maximise constraint placed on the model (a vector of all 0s produces no objective function and thus creates a feasibility test). We present an example where our objective function might be to maximise:

$2 \times Q1 + 1 \times Q$

Thus the $.obj$ vector reads:

2 1 0 0

In addition, a single variable $.ModelSense$ is included to indicate if a model is to maximised or minimised, in our case:

'max'

Thus we present a complete overview of the required components of an ILP/MLP model generated in Matlab using the Gurobi API.

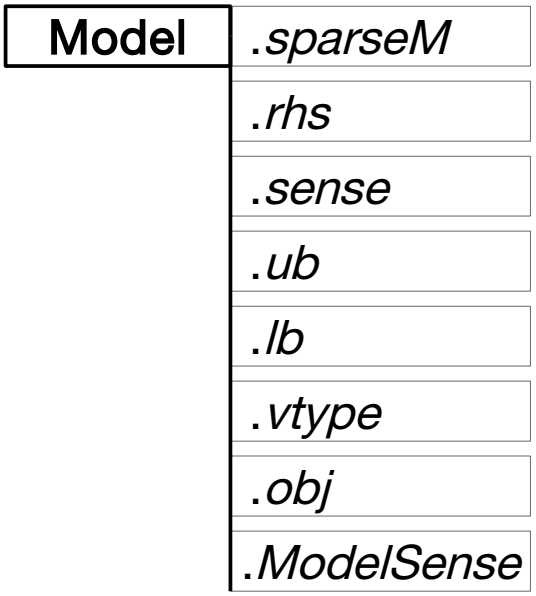


Figure 5.19: The model including all mandatory vectors and matrices.

Additionally we list the simple model which can be produced from the matrices and vectors defined above.

```
Maximize
 2 Q1 + Q2
Subject To
 Q1 + Q2 >= 1
 Q3 + Q4 >= 1
Bounds
 Q1 <= 1
 Q2 <= 1
 Q3 <= 1
 Q4 <= 1
```

```
Generals
 Q1  Q2  Q3  Q4
End
```

There are a number of additional features which may be added, such as a vector specifying variable names if presentable model output is required. Typically, this model is executed from Matlab passing the constraints directly to the Gurobi solver. The result is returned, if successful the schedule is returned, along with runtime data and additional information. In Appendix B we attach a ILP model based on the standard multi-cycle model generation presented in Chapter 3. This model is presented in the .lp format, it is based on a task set of size 40 to be scheduled on 4 cores over 4 minor cycles. To give an indication of scale the sparse matrix required to represent this problem is 126 by 644 cells, with the additional vectors we described above being of length 644 or 126 depending on if they scale by the number of variables or number of constraints.

5.3.1 Model Generation Facilitating Further Optimisation

The model generation described above allows for the rapid production of ILP models and the easy introduction of new optimisation goals. If a new optimisation requires no new variables, then simply altering the obj vector to include the correct values would suffice. Additional constraints and even new variables can easily be added to facilitate optimisations which require them. Once updated, the model generation tool is capable of rapidly re-producing the same constraints for task sets with different parameters.

While the number of possible optimisations is vast, a few examples could include:

- An optimisation which seeks to maximise the minimum spare capacity available on each core during each minor cycle. Our current optimisation in Section 5.2 seeks to maximise capacity in a simplistic way, not dictating where this capacity is required. A more complex version of this optimisation could seek to ensure that a minimum amount of spare capacity is guaranteed in all locations.

- Optimisation could be implemented alongside the task splitting approach from Chapter 4. The model may be extended to include a means of accounting for context switching overheads. The optimisation might then be to maximise capacity in a particular criticality level while ensuring that any overheads of splitting tasks do not mitigate the gains in schedulability.
- Following from the note on task ordering in Section 3.2.2 an optimisation could be performed to, where possible, ensure that higher importance tasks are scheduled earlier in a minor cycle. The aim is to reduce the likelihood of the higher importance tasks not executing in the case of a criticality overrun.

The automated production of ILP/MLP models allows for system parameters to be tweaked and allocations tested in a convenient manner.

5.3.2 Scalability

While our experiments across chapters 3 4 and 5 scale up parameters to 100 tasks, 12 cores, 5 criticality levels, these values do not represent the maximum values that the solver can scale too and solve within a reasonable time. These limits were chosen due to the increase in cost of generating the task sets, rather than actually running them. During the setup and testing of the feasibility tests in Chapters 3 and 4, initial investigations suggest that task sets with 500 or more tasks scheduled on 32 cores over 5+ criticality levels solved in less than 1 second. The performance when optimisation is taken into account is less predictable, however initial investigation did indicate parameters could be increased without a significant increase in the solvers execution time. In all, our scalability experiments across all Chapters 3, 4 and 5 were limited by the time taken to generate a sufficient number of sample task sets, rather than any limits found with the solver.

5.3.3 Unexpected Timing Behaviour

One of the downsides to utilising a commercial Linear Programming solver is the multitude of methods it may employ and their unpredictability. For example, one task set (containing 20 tasks, 2 criticality levels over 4 cores) took well over 1 minute to solve when the solver was provided with 8 cores. However, when given

4 cores it came to a solution in just a fraction of a second. In addition, any tweaks to the model itself, be it criticality levels, tasks, cores, distribution of criticality levels amongst tasks or the difference between WCETs are each criticality level can have an impact (positive or negative) on the run-time of the model. In general our models solve extremely quickly, however there remains an element of unpredictability largely down to the black box solver and the large number of parameters.

5.4 Summary

This chapter has illustrated how a mixed criticality cyclic executive schedule may be optimised to suit a particular need or design goal. We investigated how the number of cores utilised by HI criticality tasks may be reduced to minimise verification costs. In addition we considered how the capacity either side of the barrier, both the HI and LO modes, may be maximised to aid design, schedule additional tasks or to account for tasks with potentially unbounded WCETs.

In this chapter we have explored how optimisation can be used as part of our ILP/MLP based modelling. We illustrated two different optimisations and evaluated their effectiveness:

1. The first optimisation attempts to schedule HI criticality tasks on as few cores as possible in order to reduce the verification and certification costs associated with verifying a multi-core application. We then allow lower criticality work to make use of the entire platform. This optimisation hopes to find a means of supporting *near future* mixed criticality systems where verifying HI criticality work on a large number of cores is not feasible[1].

2. Our second optimisation draws on the notion that some tasks may have potentially unbounded execution times. As such, if we can maximise the spare capacity within their criticality level, that task may be able to provide an increased quality of service. In addition, in the LO criticality case increasing the spare capacity may help reduce the likelihood of tasks being suspended after

[1] It might not be feasible if you are limited by platform resources and the increased pessimism of scheduling HI criticality work on multiple cores is very large.

a criticality change. In the HI criticality case additional spare capacity effectively increases the time at which the synchronised criticality switch occurs, this will reduce the chance of an overrun being triggered if a task executes a little beyond its LO WCET prediction.

Overall, we aim to illustrate that there are many possible optimisations that could be applied to our model. These examples are intended to demonstrate some of the aspects that could be targeted for optimisation.

Chapter 6

Case Study

While randomly generated task sets are able to reasonably capture and assess the performance of our techniques, the application of a *real world* benchmark is a valuable addition. The generated task sets are based on our assumptions of how a mixed criticality system is composed and what requirements it might have. During the course of this work we were provided with an example model by an industrial partner, BAE Systems[1]. This model provides a representative industrial case study, based on an avionics system, allowing us to investigate the performance of our allocation technique on a *real world* example. This chapter focuses on applying much of the work in the prior chapters to this one specific example. We begin by examining the set of tasks provided, and discuss its significant features and how it is adapted slightly to suit our needs. We apply our ILP task allocation and MLP task splitting and optimisation techniques and discuss the resulting schedules. Finally, we describe some initial work performed with the case study which applies speed-up factor theory to investigate its schedulability on a uni-processor platform.

6.1 Case Study Overview

The model provided contains information on a system with varying levels of criticality in the avionics domain. While the model was obfuscated away from the real system, it remained representative. The model we were provided with contained 20 tasks, these are shown in Table 6.1.

[1] http://www.baesystems.com

	C(LO)	C(HI)	T/D	L
I/O_1	3.6	4.5	20	HI
I/O_2	0.8	1	20	HI
I/O_3	0.8	1	20	HI
I/O_4	4.8	6	40	HI
I/O_5	4.8	6	40	HI
I/O_6	4.8	6	40	HI
I/O_7	1.6	2	20	HI
I/O_8	0.4	0.5	40	HI
I/O_9	0.2	0.25	80	HI
P_1	1.2	1.5	20	HI
P_2	0.4	0.5	20	HI
P_3	0.8	1	20	HI
P_4	3.2	4	20	HI
P_5	1.6	2	20	HI
P_6	2.4	3	20	HI
PL_1	6	6	20	LO
PL_2	3	3	20	LO
PL_3	20	20	80	LO
I/OL_1	17	17	40	LO
SYS	0.25	0.25	40	LO

Table 6.1: Case Study: Complete task set.

The periods that were initially in Hz have been converted to ms. In addition, the period of I/O_9 was originally equal to 100, however, we reduced this to 80 to fit our cyclic executive structure. This reduction ensures the task executes once per major cycle and given its very small execution times will have little to no impact on overall schedulability. There are a number of points to note about the model:

- The initial task set, while assigned HI and LO criticality levels, did not have a WCET for each criticality level ($C(LO)/C(HI)$). After discussion with the industrial partner we concluded that the initial given values would be used for $C(HI)$, while the LO values were derived by $C(LO) = C(HI) \times 0.8$.

- The HI and LO criticality levels can be considered equivalent to DAL levels (Design Assurance Levels[2]) B and D respectively.

- While the task set contains a large number of HI criticality tasks, the LO criticality tasks have some of the largest WCETs.

[2]https://www.rtca.org/, http://www.eurocae.net/

- The SYS is a system management task designed to monitor all tasks in the system.

- As evident from its overall utilisation, the system was designed to run on a 3 core platform. The use of such an architecture is consistent with other safety critical domains which might execute on TriCore platforms such as Infineon[3].

- We observe two way communication between LO and HI criticality tasks with the data values being transmitted represented in bytes.

- The platform was designed to execute on a cyclic executive based scheduler, this is also clear from the harmonic periods 20, 40 and 80 making cyclic executive scheduling easier.

- Tasks PL_3 & I/O_1 represent a subsystem which may improve its quality of service with any additional computational capacity. This provided the inspiration for the work in Chapter 5.2.

In addition to the task set in Table 6.1, we present a diagram detailing the system in Figure 6.1. This diagram represents each task from Table 6.1, as a circle where its name, period and WCET are listed. In addition, the diagram illustrates the communication between tasks, the values on each arc indicate the size of the data transmission in bytes. While information is provided regarding communication, for our work we consider each task as independent. HI criticality tasks are indicated by the circles filled in green, while LO are left empty.

We apply the techniques developed in Chapters 3,4 and 5 to this case study in the following section.

[3]https://www.infineon.com

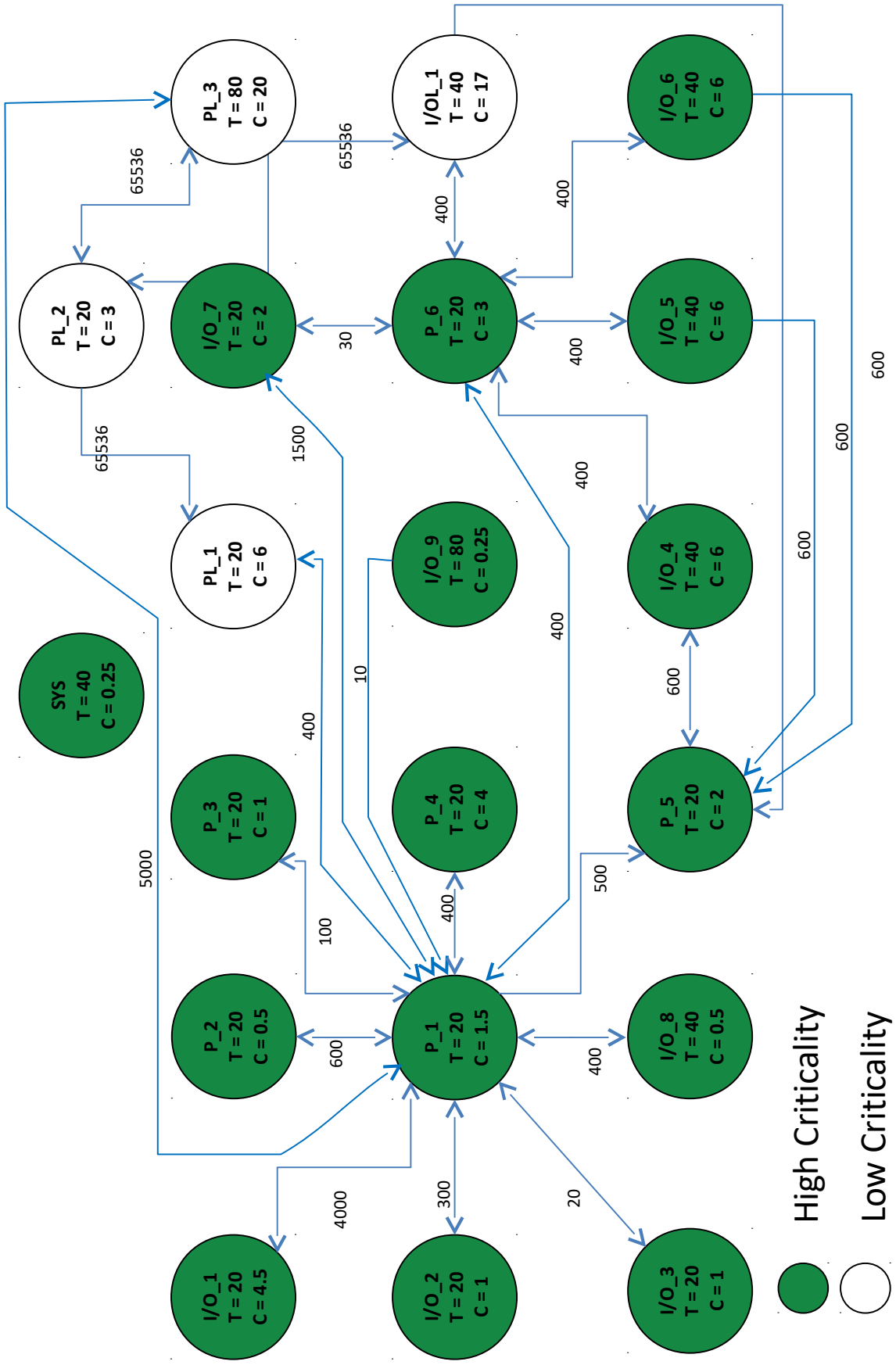


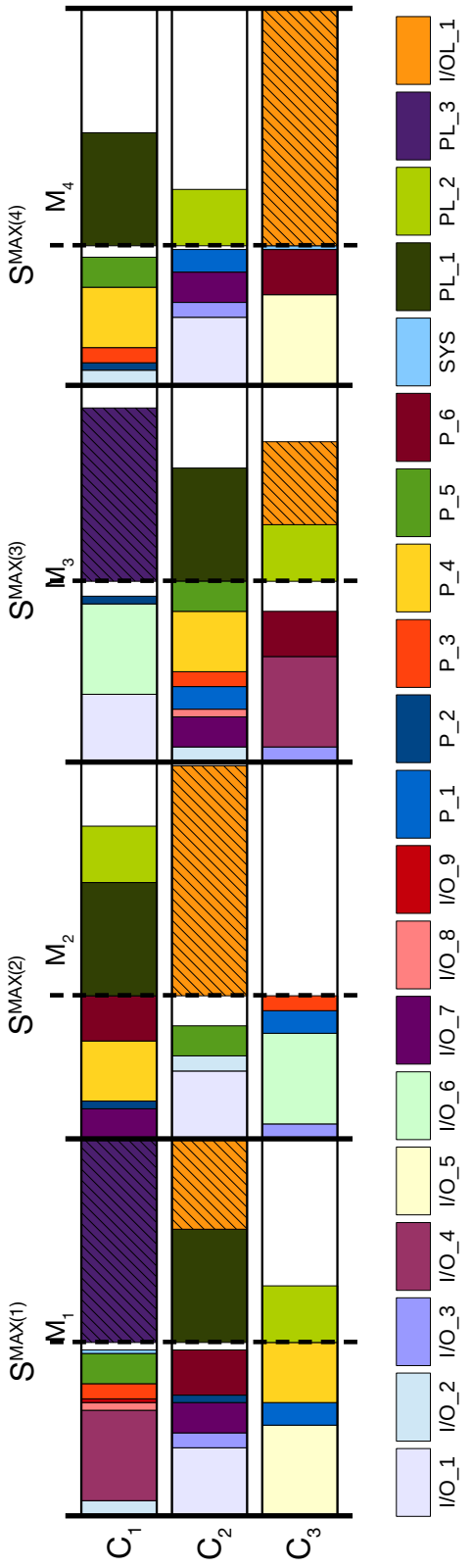
Figure 6.1: A representation of the system model provided by BAE Systems.

6.2 Modelling using Linear Programming

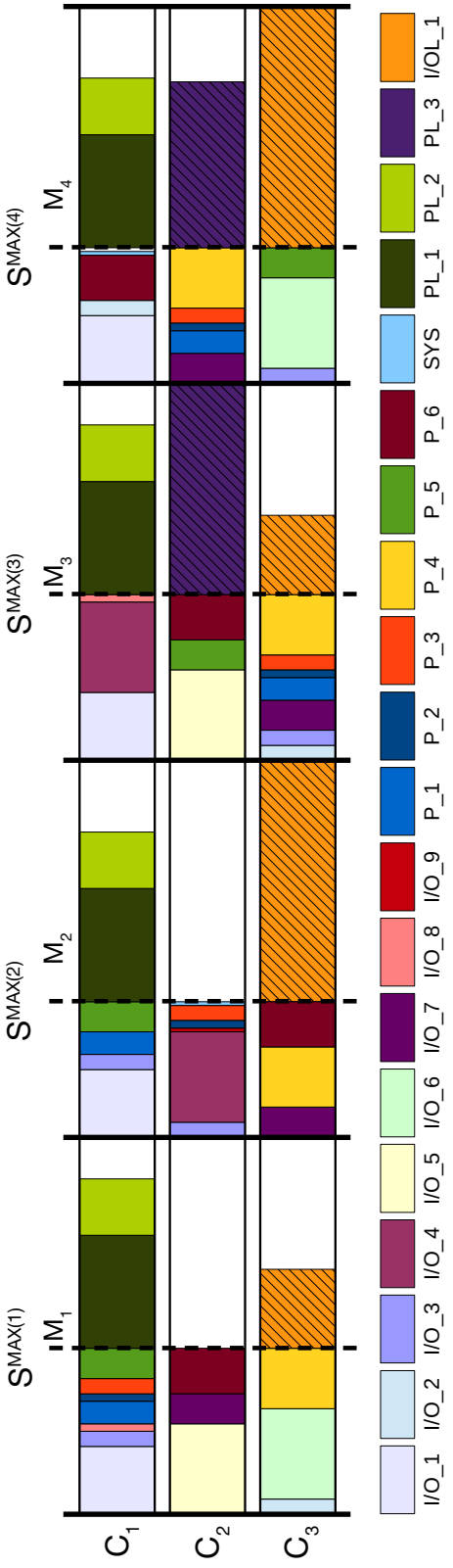
In order to allocate the case study to our mixed criticality cyclic executive model we must define the parameters of the CE. Conveniently, the case study was based on a cyclic executive type architecture, thus all periods are harmonic and it is easy to define the minor and major cycle length. The minor cycle length was set at 20, $T^F = 20$, and the major cycle length was set at 80, $T^M = 80$, giving a system where four different minor cycles run per major cycle. Tasks have periods of either 20, 40 or 80 running 4 times, twice, or once per major cycle respectively.

We applied the ILP/MLP modelling and allocation techniques to the model. At first glance, it is very clear that while the original system was designed to execute on 3 cores, our platform, which makes use of synchronised criticality switching, would not be able to achieve this. This is due to tasks PL_3 and I/OL_1, both having WCETs equal to, or very close to the duration of the minor cycle, as some HI criticality work must execute every cycle, there is no platform that could schedule these tasks. As such we immediately turn to the work in Chapter 4 and allow tasks PL_3 and I/OL_1 to split, the resulting schedule easily fits on 3 cores and is illustrated in Figure 6.2a. This illustrates a schedule which splits I/OL_1 over all 4 minor cycles (as it executes once every two cycles) and PL_3 over minor cycles 1 and 3 (splitting is illustrated by the hatched area). Each 'box' separated by the longer solid vertical lines represents a core within a minor cycle. Within each minor cycle is a dashed line mid cycle running across all cores, this represents the predicted invocation of the barrier protocol. Beyond these necessary splits, we see a good deal of spare capacity, indicated in the schedule as the empty 'white' space.

Following discussion with the industrial partner we established that some tasks, in particular the LO criticality tasks with the large WCET values may benefit from any spare capacity and have potentially unbounded execution times. These tasks are able to continue execution where possible to provide an improved quality of service for the application, an example might be an increased refresh rate for a display. This motivated the optimisation presented in Chapter 5 which seeks to maximise the potential spare capacity in a given criticality level. In Figure 6.2b we apply this technique to the current model and seek to maximise the LO criticality capacity to benefit the tasks with potentially unbounded execution times.



(a) The Case Study on a 3 core platform with PL_3 & I/O_1 split by the solver.



(b) The Case Study on a 3 core platform with PL_3 & I/O_1 split by the solver and the schedule is optimised to maximise spare LO criticality capacity.

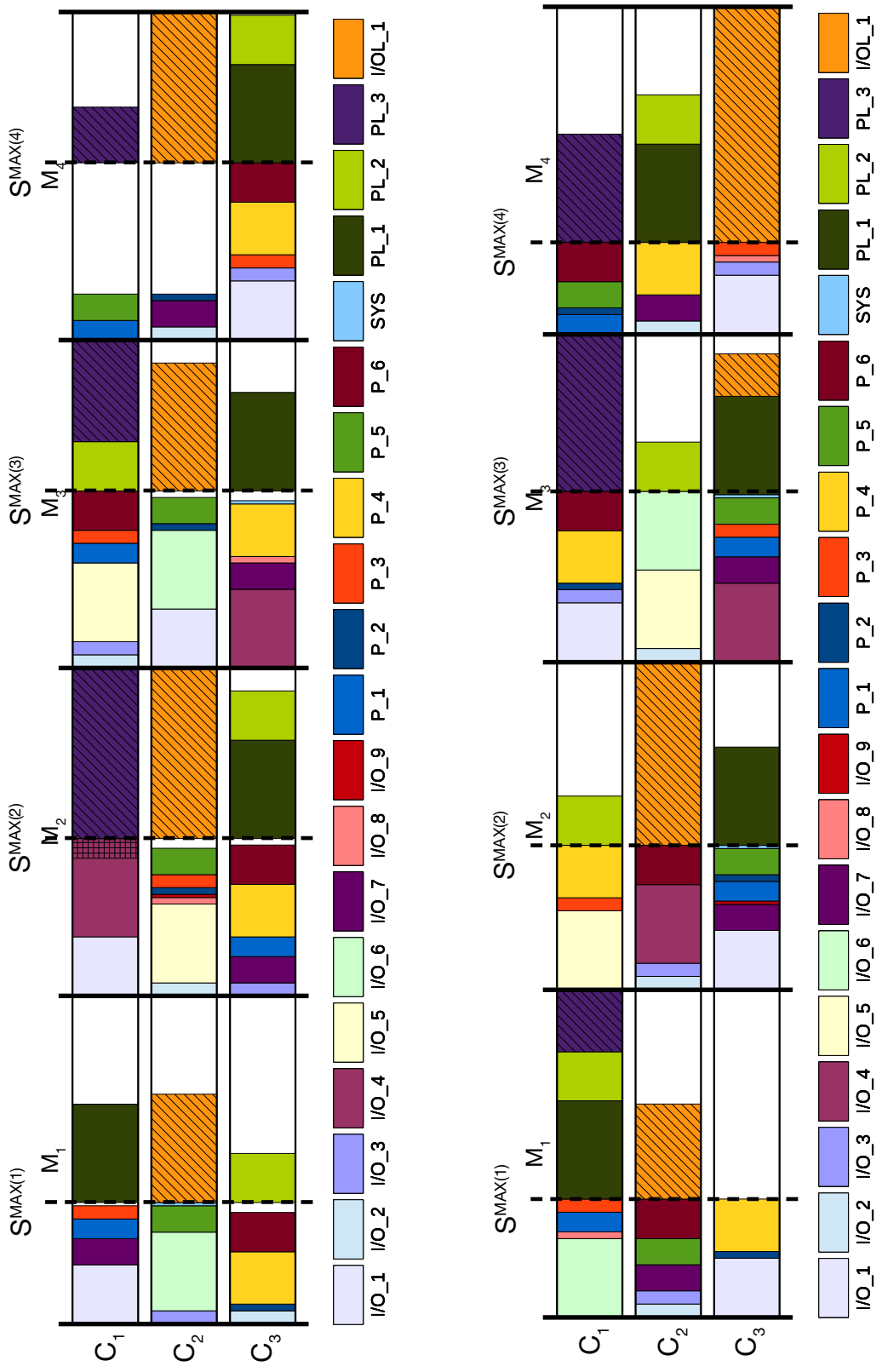
Figure 6.2: 3 core, LO capacity maximisation with LO criticality task splitting.

It is visibly clear that very little spare capacity (white space) exists to the left of the barrier in the optimised schedule (Figure 6.3b) when compared to the unoptimised Figure 6.2a.

In keeping with the work on task splitting, we also present a schedule where HI criticality task I/O_4 is allowed to split. The non-optimised schedule is shown in Figure 6.3a and the optimised schedule (to maximise LO criticality capacity) is illustrated in Figure 6.3b. Crucially here, although I/O_4 is permitted to split, it does not, as the solver does not need it to split to become schedulable and no additional capacity can be gained. However, the resulting schedule differs from that in Figures 6.2a and 6.2b, this highlights how even if additional constraints do not split tasks, they influence the solver in some way, both however, provide valid schedules. This is indicative of the behaviour of the solver, different sets of constraints often lead to the creation of different initial solutions, these different initial solutions follow alternative iterations toward the optimised goal. It is always possible, or even likely, that multiple optimised solutions exist.

One of the features of HI criticality task splitting is the ability for the solver to allocate some of the HI criticality execution to the LO mode with the goal of reducing the likelihood of a criticality change where additional capacity is available. The non-optimised schedule shows I/O_4 doing just that in minor cycle 2, the horizontally hatched area representing an extension to the LO WCET (angled hatching represents split tasks). Given I/O_4 has a WCET made up of two containers $C_{I/O\_4}(LO)$ and $C_{I/O\_4}(EX)$ and has a LO execution time equal to $C(LO) = 4.8$ [4] and a HI execution time equal to $C(HI) = 6$, thus the EX time required to go from $C(LO)$ to $C(HI)$ is $C(EX) = 1.2$. As such, a *standard* distribution of execution time between LO and EX could expect to be $C_{I/O\_4}(LO) = 4.8$ and $C_{I/O\_4}(EX) = 1.2$. However, in the schedule in Figure 6.3b $C_{I/O\_4}(LO) = 6$ and $C_{I/O\_4}(EX) = 0$. This movement of execution time was performed by the solver in order to better meet the optimisation goal. In addition, this functionality may become the goal of optimisation itself, as the solver may seek to maximise the amount of EX execution time allocated to the LO containers, thus, in theory, reducing the likelihood of a criticality overrun.

[4] While the time here is in decimal form, when implemented in the model all execution times are multiplied such that non are decimal in order to maintain integer units of time.



(a) The Case Study on a 3 core platform with PL_3, I/O_1 & I/O_4 split by the solver.

(b) The Case Study on a 3 core platform with PL_3, I/O_1 & I/O_4 split by the solver and the schedule is optimised to maximise spare LO criticality capacity.

Figure 6.3: 3 core, LO capacity maximisation with LO & HI criticality task splitting.

We then applied the first optimisation presented in Chapter 5 to the case study. This optimisation seeks to minimise the number of cores used by the HI criticality tasks. Given the large number of HI criticality tasks this appeared to be a challenging optimisation to illustrate properly with this model. While we observe spare capacity, the large number of HI criticality tasks makes it difficult to see how the number of cores used by HI criticality work could be reduced (on the current 3 core platform). However, we begin with the 3 core example to investigate the schedules the solver would produce.

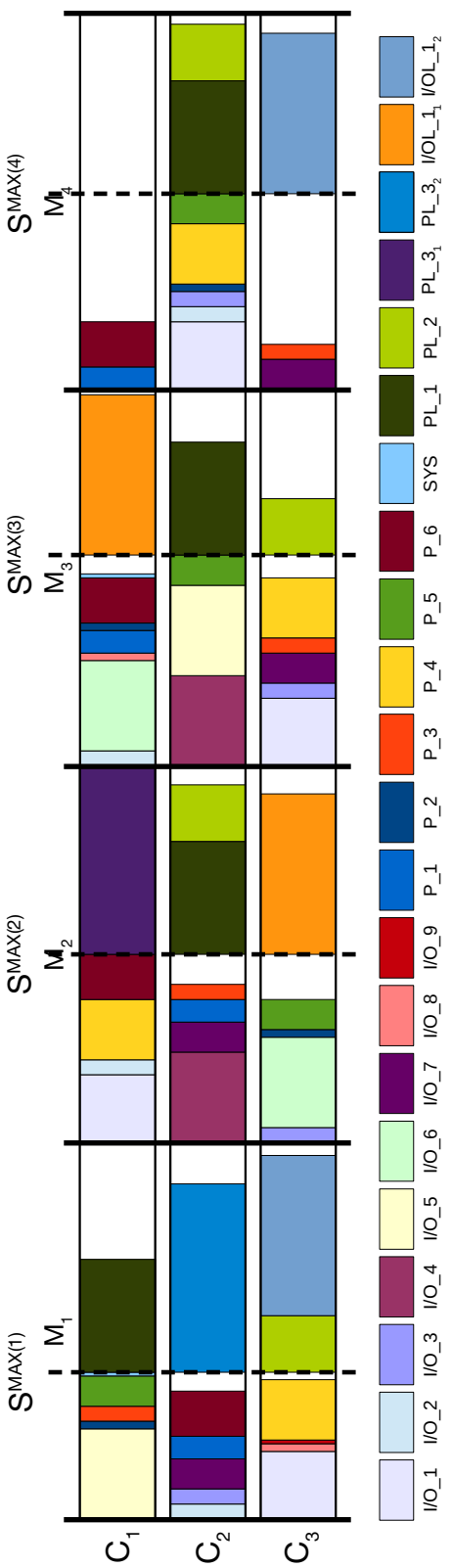
When optimising we do not permit task splitting, this was limited to the investigation in Chapter 4. Thus to make the set schedulable we altered tasks PL_3 and I/OL_1 as these tasks have prohibitively large WCETs. We statically split tasks PL_3 and I/OL_1 to provide a schedulable system without needing the tool to find a suitable split. The additional tasks are listed in Table 6.2 which works as an addition to Table 6.1 replacing tasks PL_3 and I/OL_1.

	C(LO)	C(HI)	T/D	L
PL_3_1	10	10	80	LO
PL_3_2	10	10	80	LO
I/OL_1_1	8.5	8.5	40	LO
I/OL_1_2	8.5	8.5	40	LO

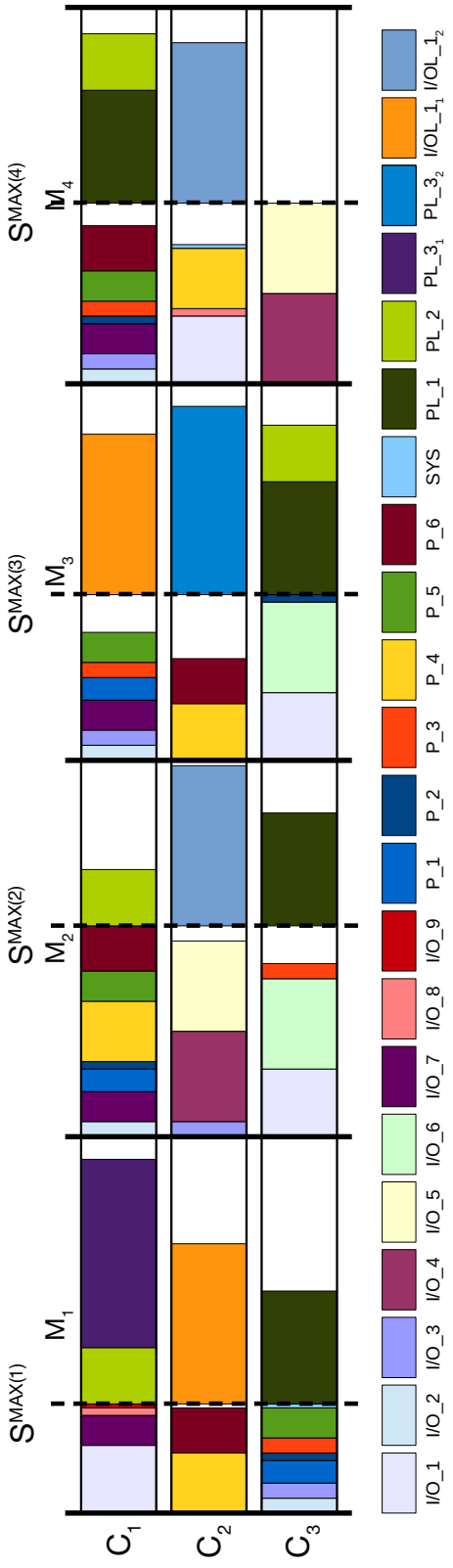
Table 6.2: Case Study: Extensions to 6.1.

Firstly, with a static split, we may perform a simple allocation of tasks on a 3 core system, with no optimisation or tool based splitting. The resulting allocation is shown in Figure 6.4a. As we have split PL_3 and I/OL_1 into manageable sub tasks this allocation was easy for the solver to perform using a standard ILP allocation model. The schedule illustrates a reasonable amount of spare capacity, however it seems unlikely that the number of cores used by HI criticality tasks could be reduced.

We then implemented the optimisation to minimise the number of cores used by high criticality tasks, the schedule is shown in Figure 6.4b. Firstly, it is immediately clear that the solver has not managed to reduce the number of cores used. We again observe a different schedule from that in Figure 6.4a as when the solver has a different goal, different solutions are produced.



(a) The Case Study on 3 cores with statically split tasks.



(b) The Case Study on 3 cores with statically split tasks, optimised to reduce the number of cores used by HI criticality tasks.

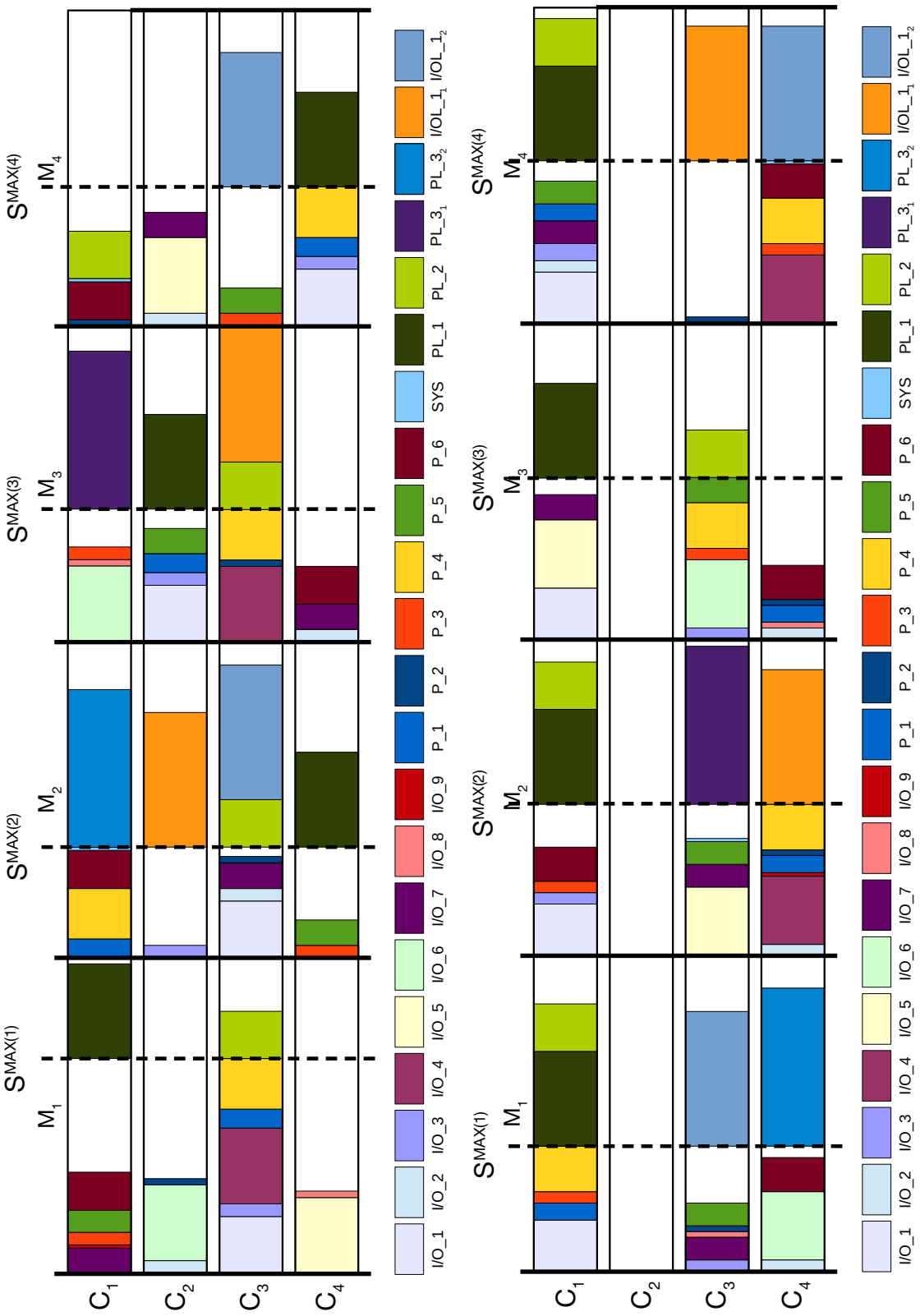
Figure 6.4: 3 core, statically split with HI criticality capacity maximised.

It is clear from Figure 6.4b that while some spare capacity exists, it is not enough to allow HI criticality tasks to execute on only 2 cores. With this in mind we move to a 4 core platform, the non-optimised 'standard' allocation is shown in Figure 6.5a. It is immediately apparent that there is a large amount of spare capacity. This step is designed to illustrate how the optimisation can be applied to this model. It follows that, as we already know the system is schedulable on 3 cores, it must be possible to schedule it on 4. In addition, we know that if it is schedulable on 3 cores, then adding an additional core and optimising to reduce the number of cores should return the schedule to a 3 core setup. With this in mind, we use this example as an illustration of functionality.

Again, we optimise to reduce the number of cores used by HI criticality tasks. Figure 6.5b illustrates how core 2 is not required by HI criticality, or any tasks in the system (as expected). Thus in the optimised schedule, no task is scheduled upon it. The resulting schedule follows the model used in Chapter 5 which allows for a 10% reduction in a HI criticality task's WCET for each core not used by any HI criticality task. Again, we know that the system is 3 core schedulable, and there will be no schedulability gain by reducing the WCETs by 10%, we simply perform this step to illustrate how these techniques can be applied to the case study.

Task splitting in some form, static or dynamic (by the solver), was required in order to schedule the task set on its original 3 cores. The first optimisation was inspired by the case study and its tasks with potentially unbounded execution times. We illustrated how this optimisation can be applied to the example system. Finally, we applied the optimisation to reduce the number of cores HI criticality work is executing upon. While this optimisation was not directly applicable to the example, due to the overall utilisation of the task set and the balance of HI and LO criticality tasks, we were still able to illustrate its use and the resulting schedules produced.

Overall, it is clear that while we are able to illustrate the performance and implementation of our allocation techniques, they are not necessarily valuable as this is a completed system. A more applicable use of our ILP/MLP models might be the addition of new functionality in the form of an additional task, the task could be added and the schedulability and possible allocation/s investigated.



(a) The Case Study on 4 cores with statically split tasks.

(b) The Case Study on 4 cores with statically split tasks, optimised to reduce the number of cores used by HI criticality tasks.

Figure 6.5: 4 core, statically split with the number of cores utilised by HI criticality tasks minimised.

6.3 Speedup Factor

In addition to applying the ILP based task allocation approaches we considered how the requirements of the case study measure up in a uni-processor fixed priority context. We began by considering the number of cores required for its execution and how this can be investigated using speed-up factors. Speedup, also known as Amdahl's argument [68], is a means of assessing how fast a single processor would need to be in order to schedule a set of tasks designed for a multi-core. This section describes how this initial work developed.

We considered the example system model as the task set shown in Table 6.3, which represents the original values provided, with no $C(LO)$ or $C(HI)$ differentiation.

	C	T/D	L
I/O_1	4.5	20	HI
I/O_2	1	20	HI
I/O_3	1	20	HI
I/O_4	6	40	HI
I/O_5	6	40	HI
I/O_6	6	40	HI
I/O_7	2	20	HI
I/O_8	0.5	40	HI
I/O_9	0.25	80	HI
P_1	1.5	20	HI
P_2	0.5	20	HI
P_3	1	20	HI
P_4	4	20	HI
P_5	2	20	HI
P_6	3	20	HI
PL_1	6	20	LO
PL_2	3	20	LO
PL_3	20	80	LO
I/OL_1	17	40	LO
SYS	0.25	40	LO

Table 6.3: Case Study: Original Model (no criticality levels).

This initial task set does not include differing values for LO WCETs of HI criticality tasks. We ran this task set through the AMCrtb [15] schedulability test and reduced the original WCET values until schedulable to determine the speed-up factor.

$$S = \frac{C(Old)}{C(New)} \tag{6.1}$$

$$S(I/O\_1) = \frac{4.5}{1.71} \tag{6.2}$$

The speedup factor S is calculated by dividing the *Old* WCET (originally from Table 6.3) by a new reduced value. All tasks are reduced by the same % until schedulable, once schedulable the speedup factor can be determined. The result is a speed-up factor of 2.63, given that this system is designed to run on three 1.2 Ghz CPUs we can calculate the frequency required of a single core to run the system.

$$1.2 \times 2.63 = 3.156 \tag{6.3}$$

In this case the system would require a CPU running at 3.16 Ghz to schedule on a single processor.

As we are dealing with a mixed criticality system, the HI criticality tasks should be given a LO WCET derived from the provided HI WCETs. As this reduction was not defined for us (although an estimate of 80% was confirmed to be reasonable) we began investigating the impact of reducing the LO WCET as a % of the HI to attempt to reduce the speedup factor. Given the task set in Table 6.1 which provides LO WCET values for HI criticality tasks calculated as 80% of their HI WCETs, we apply the same speedup factor technique. We reduce the WCETs (both LO and HI) of each task by the same % until the system is schedulable. The resulting speedup factor is 2.33 which required a 2.79Ghz uni-processor to schedule all tasks.

As well as considering the case above, we also considered the speedup factor when the LO tasks were assumed to be 90%, 70% , 60% and 50% of the original WCET. The resulting speedup factors and the required speed of the CPU to schedule the task set are shown in Table 6.4.

LO % of HI	100%	90%	80%	70%	60%	50%
Speedup Factor	2.63	2.5	2.33	2.17	2.04	1.89
Req CPU Speed (GHz)	3.16	3	2.79	2.61	2.45	2.26

Table 6.4: Case Study: Speedup Factors.

In addition, the results from Table 6.4 as plotted in Figure 6.6:

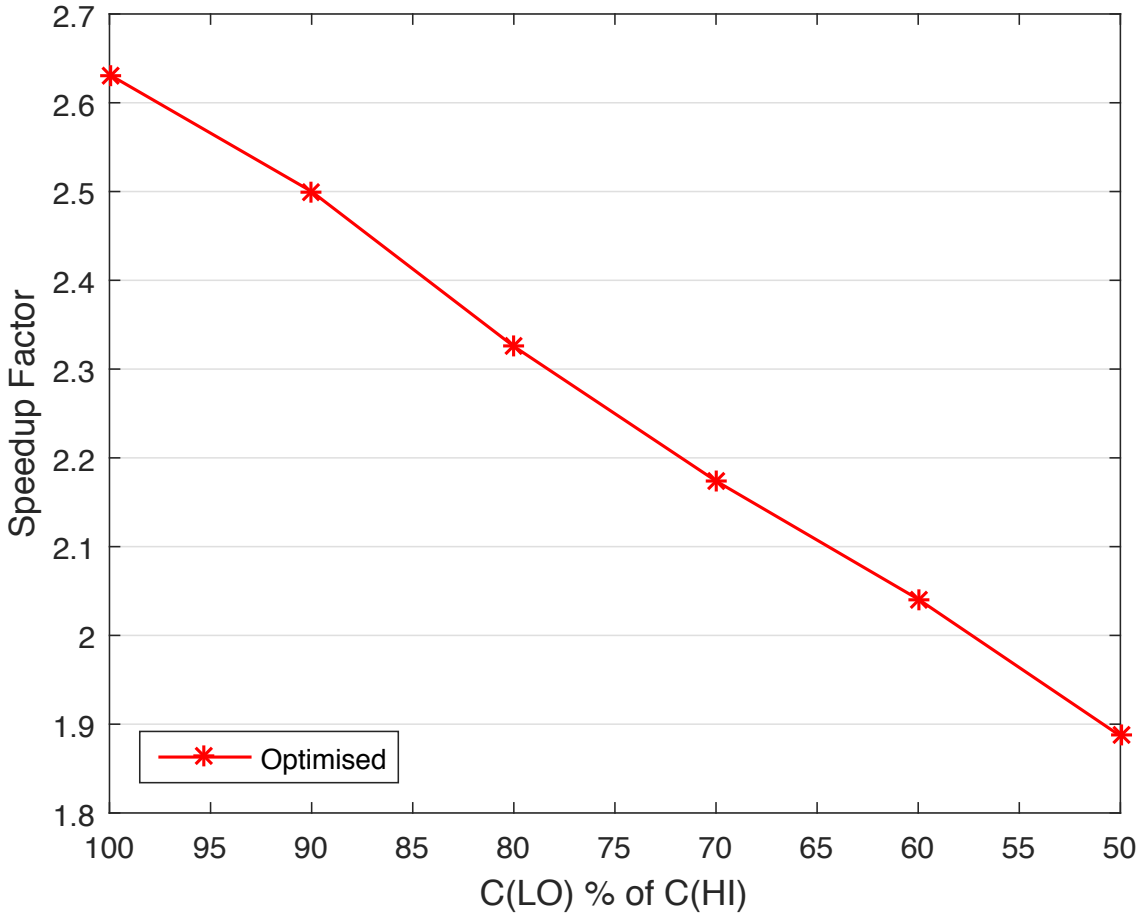


Figure 6.6: Speedup Graph.

We illustrate here how mixed criticality functionality makes a big difference in the overall average system load.

This analysis presents a simplified look at the schedulability of the system on a single processor. We can make two main points. Firstly, this work shows that by providing smaller LO criticality WCETs for HI criticality tasks it is possible to reduce the speedup factor by a significant amount (0.3 from 100% to 80%). Secondly, it is clear that the clock speed required makes it infeasible to schedule this system on a single processor in an embedded real-time scenario. Such frequencies are beyond those provided by typical embedded hardware due to high power draw and

thermal output. As such we concluded that a multi-processor approach is required.

6.4 Summary

This chapter has described and examined the mixed criticality case study provided by BAE Systems[5]. We have discussed its features and how they are both similar and different from our perception of a mixed criticality system. We have applied our ILP/MLP based task allocation techniques to schedule the case study on a multi-core cyclic executive platform. We applied task splitting to overcome a number of tasks with very large execution times. Additionally, we coupled this with two optimisations; we optimised the task set to maximise the spare capacity within the LO criticality mode and we optimised the system to reduce the number of cores used by HI criticality tasks. Finally, we make brief mention of some initial work investigating how speedup factor can be used to gauge the requirements of this system if it were scheduled on a uni-processor platform.

The ability to apply our tools and techniques to a *real world* example case study is an invaluable step. Not only do we establish how our tools perform given genuine task data, but we also gain understanding as to how they may be best used to support future expansion of the system.

[5]http://www.baesystems.com

Chapter 7

Conclusion

7.1 A return to the thesis hypothesis

We return to the thesis hypothesis:

Mixed Criticality Cyclic Executives provide an attractive platform for highly critical near-future systems. The challenge of allocating tasks to the platform, providing support for design and aiding in allocation optimisation can be achieved through the use of Linear Programming.

We have illustrated how merging the deterministic Cyclic Executive with the flexibility of mixed criticality functionality provides an attractive platform for near future mixed criticality applications. Fundamentally, we retain a familiar scheduling paradigm and extend it to encompass additional functionality, rather than starting from scratch. As such legacy applications designed to run on cyclic executive systems could be ported to a mixed criticality cyclic executive with minimal difficulty. Meanwhile, newer applications, perhaps those able to utilise multi-threaded workloads, are able to execute alongside due to the complete separation provided between criticality levels. Proving the isolation of higher criticality work is simple as it always executes before lower criticality work. While the classic problems of cyclic executives are not removed with this work, the mixed criticality approach allows for greater leverage of the platform resources under typical execution.

Throughout this work we made extensive use of linear programming tools to aid

in the allocation of tasks to mixed criticality cyclic executives. The tools provided in both the standard allocation and the more complex splitting and optimisation functionality are intended as a means to aid system design. It may be best used as part of a design process, allowing the system designer to quickly discover how to adjust the parameters of their system, or to discover where additional capacity might lie to introduce more functionality. Linear Programming optimisation provides a large number of parameters which may be adjusted to allow the system designer to ask *questions* about the allocation and design of their system.

7.2 Results of this thesis

The main results reported in this thesis are summarised as follows:

1. In Chapter 3 we describe the process of allocating tasks to a mixed criticality cyclic executive on multiple cores. We began by describing how heuristic based approaches could be applied to a simplified single cycle ($T^F = T^M$) case. Furthermore, we compared these heuristics against an optimal (in the sense that if a feasible schedule exists it will find it) Integer Linear Programming formulation. The fundamental result of this section is that for the limited single cycle case, WF (Worst Fit) and FFBB (First Fit with Branch & Bound) perform well when compared with the optimal ILP model in terms of schedulability (First Fit performs poorly due to a clash between its allocation technique and the use of the barrier).

2. The second half of Chapter 3 extends the limited single cycle model from the first half to a full cyclic executive with multiple minor cycles per major cycle. The work describes adaptations and additions to the ILP models required to support multiple minor cycles. The heuristic Worst Fit is shown to perform poorly in this case when compared with the optimal ILP formulation. In addition, experimental timing data illustrates how the ILP models find a feasible solution in a relatively short amount of time. This leads to the conclusion that when allocating a mixed criticality cyclic executive, there seems to be no reason not to apply an optimal Integer Linear Programming based approach.

3. Chapter 4 describes the process and provisions for allowing solver-driven task splitting. The first half discusses the splitting of the LO criticality tasks. We describe how splitting can be achieved by changing variables that were previously defined as integer, to continuous. We also describe additional constraints required to maintain this splitting and ensure that splitting occurs across minor cycles, not cores.

4. The second half of Chapter 4 discusses the more complex splitting of HI criticality tasks. We set out a container-based splitting approach, which tackles the problem of splitting a task with multiple WCETs. Constraints for the ILP/MLP formulation are presented to enforce this technique. The splitting described in this chapter is emphasised as *limited*, which tasks are allowed to split is up to the system designer. It is intended as a tool to facilitate design, it might allow a task with a WCET close to or greater than T^F to be split and scheduled, or it might even out capacity across all minor cycles to allow an additional task to be scheduled. An experimental investigation considers the schedulability gains from task splitting to a variety of degrees and reports fast solution execution times.

5. In Chapter 5 we consider how optimisation can be used to influence the allocation of tasks to mixed criticality cyclic executives. The first optimisation presented aims to minimise the number of cores utilised by HI criticality tasks. This stems from the notion that certification and verification on multiple cores adds increasingly extreme levels of pessimism. The fewer cores used by tasks which required stringent verification the better. We illustrate the result in this reduction of cores by reducing the WCET of each task by 10% for each core unused by all HI criticality tasks. This, perhaps pessimistically, is designed to model the reduced complexity in verification and thus the reduction in overall pessimism. Examples are used to illustrate the approach and experimental results show a slight gain in schedulability due to the 10% WCET reductions applied when a core is not used by HI criticality tasks.

6. The second half of Chapter 5 presents an optimisation which seeks to maximise the capacity either side of the barrier, in a particular criticality level. The

motivation for this optimisation comes from an industrial partner noting that some of their tasks have potentially unbounded execution times. As such, given additional processing capacity these tasks are able to continue executing to improve the quality of service provided by the application. Additionally this optimisation may be used to maximise the spare LO criticality capacity which in turn may help reduce the likelihood of these tasks not being executed.

7. In Chapter 6 we apply each of the techniques set out in the thesis to an example mission system case study provided by BAE Systems. We discuss the features of the given task set and what adjustments/assumptions we make to fit within our work. We apply standard task allocation, task splitting to deal with some particularly large LO criticality WCETs and finally optimisation to illustrate their effect on a real-world example. In addition we consider how the case study might be scheduled on a single processor with fixed priority scheduling via the speedup factor metric.

7.3 Future Work

A number of clear paths for future work present themselves:

- The Linear Programming model may be extended to consider additional system features such as communication and inter-task dependency.

- While the tools used in this thesis served their purpose experimentally, a fully featured tool could be developed to rapidly produce and execute LP models. This tool could be used during system design, making rapid prototyping of new allocations simple.

- Future work might also extend the splitting functionality described in this thesis to allow splitting across cores and minor cycles. Cross-core splitting would need to provide constraints which ensure split sections of the same task always execute sequentially regardless of the core they are allocated to.

7.4 Final Thoughts

Throughout this work we have strongly advocated for the use of Linear Programming tools in task allocation and the desire to find scheduling policies which find a mid ground between dynamic functionality and supporting legacy software and verification processes. Linear Programming tools are often dismissed due to potentially exponential runtime costs. In reality, for the purpose of investigating static offline schedulability they are highly suitable, especially given the power of modern solvers and hardware. The ILP/MLP models proposed in this thesis could be utilised as part of a rapid design/prototyping tool which is able to regenerate a model with any changes required by the designer.

Appendices

Appendix A

Timing data for the single cycle case

We present the timing data recorded for the single cycle experiments, Figures A.1 and A.2 show data where the number of tasks and cores are scaled respectively.

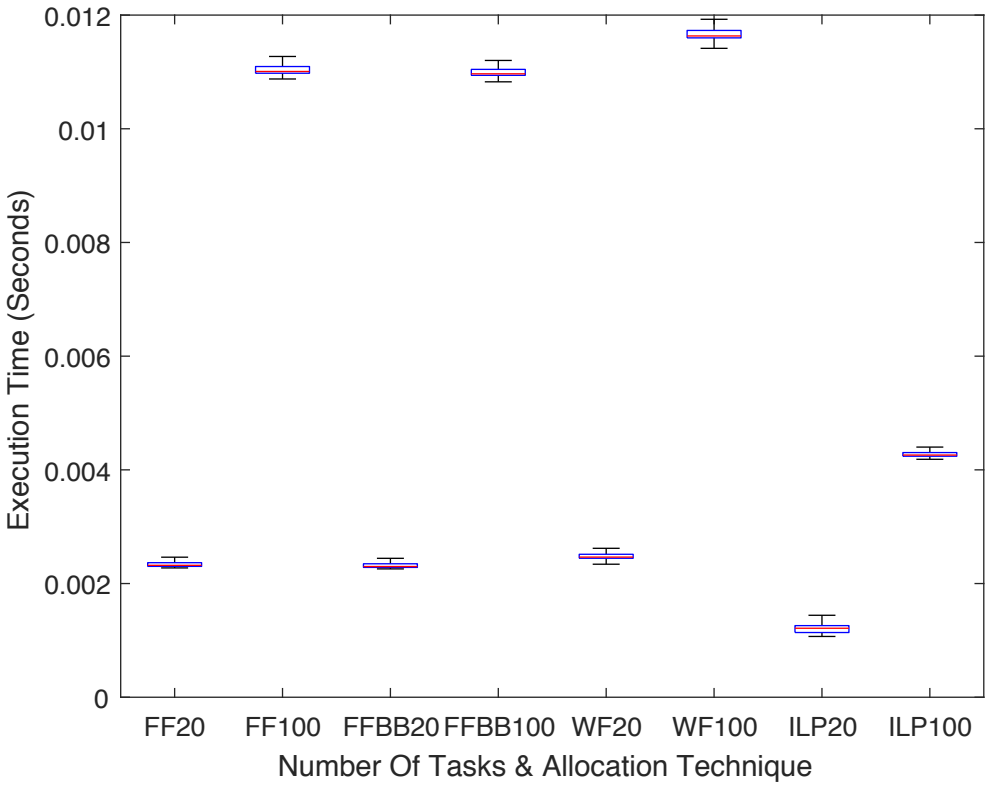


Figure A.1: A box plot illustrating the range of execution times taken for each approach as the number of tasks is increased (single cycle).

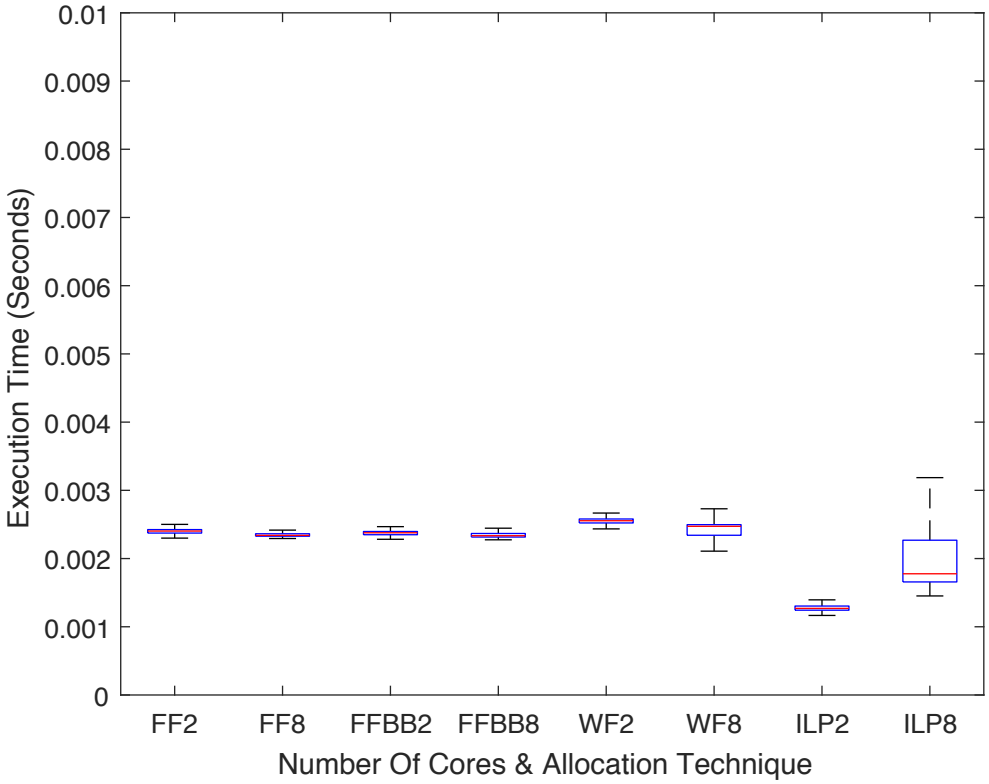


Figure A.2: A box plot illustrating the range of execution times taken for each approach as the number of cores is increased (single cycle).

We observe very fast performance in all cases, any observed variation is minor considering all approaches find solutions within a fraction of a second.

Appendix B

Full ILP model listing (using the .lp format)

```
Maximize

Subject To
Q1_11 + Q1_21 + Q1_31 + Q1_41 + Q1_12 + Q1_22 + Q1_32 + Q1_42 +
Q1_13 + Q1_23 + Q1_33 + Q1_43 + Q1_14 + Q1_24 + Q1_34 + Q1_44 = 1
Q2_11 + Q2_21 + Q2_31 + Q2_41 + Q2_12 + Q2_22 + Q2_32 + Q2_42 +
Q2_13 + Q2_23 + Q2_33 + Q2_43 + Q2_14 + Q2_24 + Q2_34 + Q2_44 = 1
Q3_11 + Q3_21 + Q3_31 + Q3_41 + Q3_12 + Q3_22 + Q3_32 + Q3_42 +
Q3_13 + Q3_23 + Q3_33 + Q3_43 + Q3_14 + Q3_24 + Q3_34 + Q3_44 = 1
Q4_11 + Q4_21 + Q4_31 + Q4_41 + Q4_12 + Q4_22 + Q4_32 + Q4_42 = 1
Q4_13 + Q4_23 + Q4_33 + Q4_43 + Q4_14 + Q4_24 + Q4_34 + Q4_44 = 1
Q5_11 + Q5_21 + Q5_31 + Q5_41 + Q5_12 + Q5_22 + Q5_32 + Q5_42 = 1
Q5_13 + Q5_23 + Q5_33 + Q5_43 + Q5_14 + Q5_24 + Q5_34 + Q5_44 = 1
Q6_11 + Q6_21 + Q6_31 + Q6_41 + Q6_12 + Q6_22 + Q6_32 + Q6_42 = 1
Q6_13 + Q6_23 + Q6_33 + Q6_43 + Q6_14 + Q6_24 + Q6_34 + Q6_44 = 1
Q7_11 + Q7_21 + Q7_31 + Q7_41 = 1
Q7_12 + Q7_22 + Q7_32 + Q7_42 = 1
Q7_13 + Q7_23 + Q7_33 + Q7_43 = 1
Q7_14 + Q7_24 + Q7_34 + Q7_44 = 1
Q8_11 + Q8_21 + Q8_31 + Q8_41 + Q8_12 + Q8_22 + Q8_32 + Q8_42 = 1
Q8_13 + Q8_23 + Q8_33 + Q8_43 + Q8_14 + Q8_24 + Q8_34 + Q8_44 = 1
Q9_11 + Q9_21 + Q9_31 + Q9_41 + Q9_12 + Q9_22 + Q9_32 + Q9_42 = 1
Q9_13 + Q9_23 + Q9_33 + Q9_43 + Q9_14 + Q9_24 + Q9_34 + Q9_44 = 1
Q10_11 + Q10_21 + Q10_31 + Q10_41 + Q10_12 + Q10_22 + Q10_32 + Q10_42 = 1
```

```
Q10_13 + Q10_23 + Q10_33 + Q10_43 + Q10_14 + Q10_24 + Q10_34 + Q10_44 = 1
Q11_11 + Q11_21 + Q11_31 + Q11_41 + Q11_12 + Q11_22 + Q11_32 +
Q11_42 + Q11_13 + Q11_23 + Q11_33 + Q11_43 + Q11_14 + Q11_24 + Q11_34 + Q11_44 = 1
Q12_11 + Q12_21 + Q12_31 + Q12_41 = 1
Q12_12 + Q12_22 + Q12_32 + Q12_42 = 1
Q12_13 + Q12_23 + Q12_33 + Q12_43 = 1
Q12_14 + Q12_24 + Q12_34 + Q12_44 = 1
Q13_11 + Q13_21 + Q13_31 + Q13_41 = 1
Q13_12 + Q13_22 + Q13_32 + Q13_42 = 1
Q13_13 + Q13_23 + Q13_33 + Q13_43 = 1
Q13_14 + Q13_24 + Q13_34 + Q13_44 = 1
Q14_11 + Q14_21 + Q14_31 + Q14_41 + Q14_12 + Q14_22 + Q14_32 + Q14_42 +
Q14_13 + Q14_23 + Q14_33 + Q14_43 + Q14_14 + Q14_24 + Q14_34 + Q14_44 = 1
Q15_11 + Q15_21 + Q15_31 + Q15_41 + Q15_12 + Q15_22 + Q15_32 + Q15_42 +
Q15_13 + Q15_23 + Q15_33 + Q15_43 + Q15_14 + Q15_24 + Q15_34 + Q15_44 = 1
Q16_11 + Q16_21 + Q16_31 + Q16_41 + Q16_12 + Q16_22 + Q16_32 + Q16_42 = 1
Q16_13 + Q16_23 + Q16_33 + Q16_43 + Q16_14 + Q16_24 + Q16_34 + Q16_44 = 1
Q17_11 + Q17_21 + Q17_31 + Q17_41 + Q17_12 + Q17_22 + Q17_32 + Q17_42 +
Q17_13 + Q17_23 + Q17_33 + Q17_43 + Q17_14 + Q17_24 + Q17_34 + Q17_44 = 1
Q18_11 + Q18_21 + Q18_31 + Q18_41 + Q18_12 + Q18_22 + Q18_32 + Q18_42 = 1
Q18_13 + Q18_23 + Q18_33 + Q18_43 + Q18_14 + Q18_24 + Q18_34 + Q18_44 = 1
Q19_11 + Q19_21 + Q19_31 + Q19_41 + Q19_12 + Q19_22 + Q19_32 + Q19_42 = 1
Q19_13 + Q19_23 + Q19_33 + Q19_43 + Q19_14 + Q19_24 + Q19_34 + Q19_44 = 1
Q20_11 + Q20_21 + Q20_31 + Q20_41 + Q20_12 + Q20_22 + Q20_32 + Q20_42 +
Q20_13 + Q20_23 + Q20_33 + Q20_43 + Q20_14 + Q20_24 + Q20_34 + Q20_44 = 1
Q21_11 + Q21_21 + Q21_31 + Q21_41 + Q21_12 + Q21_22 + Q21_32 + Q21_42 +
Q21_13 + Q21_23 + Q21_33 + Q21_43 + Q21_14 + Q21_24 + Q21_34 + Q21_44 = 1
Q22_11 + Q22_21 + Q22_31 + Q22_41 + Q22_12 + Q22_22 + Q22_32 + Q22_42 +
Q22_13 + Q22_23 + Q22_33 + Q22_43 + Q22_14 + Q22_24 + Q22_34 + Q22_44 = 1
Q23_11 + Q23_21 + Q23_31 + Q23_41 + Q23_12 + Q23_22 + Q23_32 + Q23_42 +
Q23_13 + Q23_23 + Q23_33 + Q23_43 + Q23_14 + Q23_24 + Q23_34 + Q23_44 = 1
Q24_11 + Q24_21 + Q24_31 + Q24_41 + Q24_12 + Q24_22 + Q24_32 + Q24_42 = 1
Q24_13 + Q24_23 + Q24_33 + Q24_43 + Q24_14 + Q24_24 + Q24_34 + Q24_44 = 1
Q25_11 + Q25_21 + Q25_31 + Q25_41 + Q25_12 + Q25_22 + Q25_32 + Q25_42 +
Q25_13 + Q25_23 + Q25_33 + Q25_43 + Q25_14 + Q25_24 + Q25_34 + Q25_44 = 1
Q26_11 + Q26_21 + Q26_31 + Q26_41 + Q26_12 + Q26_22 + Q26_32 + Q26_42 +
Q26_13 + Q26_23 + Q26_33 + Q26_43 + Q26_14 + Q26_24 + Q26_34 + Q26_44 = 1
Q27_11 + Q27_21 + Q27_31 + Q27_41 + Q27_12 + Q27_22 + Q27_32 + Q27_42 +
Q27_13 + Q27_23 + Q27_33 + Q27_43 + Q27_14 + Q27_24 + Q27_34 + Q27_44 = 1
```

```
Q28_11 + Q28_21 + Q28_31 + Q28_41 + Q28_12 + Q28_22 + Q28_32 + Q28_42 +
Q28_13 + Q28_23 + Q28_33 + Q28_43 + Q28_14 + Q28_24 + Q28_34 + Q28_44 = 1
Q29_11 + Q29_21 + Q29_31 + Q29_41 + Q29_12 + Q29_22 + Q29_32 + Q29_42 +
Q29_13 + Q29_23 + Q29_33 + Q29_43 + Q29_14 + Q29_24 + Q29_34 + Q29_44 = 1
Q30_11 + Q30_21 + Q30_31 + Q30_41 + Q30_12 + Q30_22 + Q30_32 + Q30_42 = 1
Q30_13 + Q30_23 + Q30_33 + Q30_43 + Q30_14 + Q30_24 + Q30_34 + Q30_44 = 1
Q31_11 + Q31_21 + Q31_31 + Q31_41 = 1
Q31_12 + Q31_22 + Q31_32 + Q31_42 = 1
Q31_13 + Q31_23 + Q31_33 + Q31_43 = 1
Q31_14 + Q31_24 + Q31_34 + Q31_44 = 1
Q32_11 + Q32_21 + Q32_31 + Q32_41 = 1
Q32_12 + Q32_22 + Q32_32 + Q32_42 = 1
Q32_13 + Q32_23 + Q32_33 + Q32_43 = 1
Q32_14 + Q32_24 + Q32_34 + Q32_44 = 1
Q33_11 + Q33_21 + Q33_31 + Q33_41 + Q33_12 + Q33_22 + Q33_32 + Q33_42 = 1
Q33_13 + Q33_23 + Q33_33 + Q33_43 + Q33_14 + Q33_24 + Q33_34 + Q33_44 = 1
Q34_11 + Q34_21 + Q34_31 + Q34_41 + Q34_12 + Q34_22 + Q34_32 + Q34_42 = 1
Q34_13 + Q34_23 + Q34_33 + Q34_43 + Q34_14 + Q34_24 + Q34_34 + Q34_44 = 1
Q35_11 + Q35_21 + Q35_31 + Q35_41 + Q35_12 + Q35_22 + Q35_32 + Q35_42  +
Q35_13 + Q35_23 + Q35_33 + Q35_43 + Q35_14 + Q35_24 + Q35_34 + Q35_44 = 1
Q36_11 + Q36_21 + Q36_31 + Q36_41 = 1
Q36_12 + Q36_22 + Q36_32 + Q36_42 = 1
Q36_13 + Q36_23 + Q36_33 + Q36_43 = 1
Q36_14 + Q36_24 + Q36_34 + Q36_44 = 1
Q37_11 + Q37_21 + Q37_31 + Q37_41 + Q37_12 + Q37_22 + Q37_32 + Q37_42 = 1
Q37_13 + Q37_23 + Q37_33 + Q37_43 + Q37_14 + Q37_24 + Q37_34 + Q37_44 = 1
Q38_11 + Q38_21 + Q38_31 + Q38_41 = 1
Q38_12 + Q38_22 + Q38_32 + Q38_42 = 1
Q38_13 + Q38_23 + Q38_33 + Q38_43 = 1
Q38_14 + Q38_24 + Q38_34 + Q38_44 = 1
Q39_11 + Q39_21 + Q39_31 + Q39_41 = 1
Q39_12 + Q39_22 + Q39_32 + Q39_42 = 1
Q39_13 + Q39_23 + Q39_33 + Q39_43 = 1
Q39_14 + Q39_24 + Q39_34 + Q39_44 = 1
Q40_11 + Q40_21 + Q40_31 + Q40_41 + Q40_12 + Q40_22 + Q40_32 + Q40_42 +
Q40_13 + Q40_23 + Q40_33 + Q40_43 + Q40_14 + Q40_24 + Q40_34 + Q40_44 = 1
98833 Q1_11 + 90526 Q2_11 + 61888 Q3_11 + 44370 Q4_11 + 23066 Q5_11
+ 18824 Q6_11 + 17037 Q7_11 + 11403 Q8_11 + 10387 Q9_11 + 10330 Q10_11
   + 5961 Q11_11 + 5538 Q12_11 + 3761 Q13_11 + 3008 Q14_11 + 2944 Q15_11
```

```
   + 2871 Q16_11 + 2238 Q17_11 + 2083 Q18_11 + 1139 Q19_11 + 757 Q20_11
   + X1 <= 250000
98833 Q1_21 + 90526 Q2_21 + 61888 Q3_21 + 44370 Q4_21 + 23066 Q5_21
   + 18824 Q6_21 + 17037 Q7_21 + 11403 Q8_21 + 10387 Q9_21 + 10330 Q10_21
   + 5961 Q11_21 + 5538 Q12_21 + 3761 Q13_21 + 3008 Q14_21 + 2944 Q15_21
   + 2871 Q16_21 + 2238 Q17_21 + 2083 Q18_21 + 1139 Q19_21 + 757 Q20_21
   + X1 <= 250000
98833 Q1_31 + 90526 Q2_31 + 61888 Q3_31 + 44370 Q4_31 + 23066 Q5_31
   + 18824 Q6_31 + 17037 Q7_31 + 11403 Q8_31 + 10387 Q9_31 + 10330 Q10_31
   + 5961 Q11_31 + 5538 Q12_31 + 3761 Q13_31 + 3008 Q14_31 + 2944 Q15_31
   + 2871 Q16_31 + 2238 Q17_31 + 2083 Q18_31 + 1139 Q19_31 + 757 Q20_31
   + X1 <= 250000
98833 Q1_41 + 90526 Q2_41 + 61888 Q3_41 + 44370 Q4_41 + 23066 Q5_41
   + 18824 Q6_41 + 17037 Q7_41 + 11403 Q8_41 + 10387 Q9_41 + 10330 Q10_41
   + 5961 Q11_41 + 5538 Q12_41 + 3761 Q13_41 + 3008 Q14_41 + 2944 Q15_41
   + 2871 Q16_41 + 2238 Q17_41 + 2083 Q18_41 + 1139 Q19_41 + 757 Q20_41
   + X1 <= 250000
98833 Q1_12 + 90526 Q2_12 + 61888 Q3_12 + 44370 Q4_12 + 23066 Q5_12
   + 18824 Q6_12 + 17037 Q7_12 + 11403 Q8_12 + 10387 Q9_12 + 10330 Q10_12
   + 5961 Q11_12 + 5538 Q12_12 + 3761 Q13_12 + 3008 Q14_12 + 2944 Q15_12
   + 2871 Q16_12 + 2238 Q17_12 + 2083 Q18_12 + 1139 Q19_12 + 757 Q20_12
   + X2 <= 250000
98833 Q1_22 + 90526 Q2_22 + 61888 Q3_22 + 44370 Q4_22 + 23066 Q5_22
   + 18824 Q6_22 + 17037 Q7_22 + 11403 Q8_22 + 10387 Q9_22 + 10330 Q10_22
   + 5961 Q11_22 + 5538 Q12_22 + 3761 Q13_22 + 3008 Q14_22 + 2944 Q15_22
   + 2871 Q16_22 + 2238 Q17_22 + 2083 Q18_22 + 1139 Q19_22 + 757 Q20_22
   + X2 <= 250000
98833 Q1_32 + 90526 Q2_32 + 61888 Q3_32 + 44370 Q4_32 + 23066 Q5_32
   + 18824 Q6_32 + 17037 Q7_32 + 11403 Q8_32 + 10387 Q9_32 + 10330 Q10_32
   + 5961 Q11_32 + 5538 Q12_32 + 3761 Q13_32 + 3008 Q14_32 + 2944 Q15_32
   + 2871 Q16_32 + 2238 Q17_32 + 2083 Q18_32 + 1139 Q19_32 + 757 Q20_32
   + X2 <= 250000
98833 Q1_42 + 90526 Q2_42 + 61888 Q3_42 + 44370 Q4_42 + 23066 Q5_42
   + 18824 Q6_42 + 17037 Q7_42 + 11403 Q8_42 + 10387 Q9_42 + 10330 Q10_42
   + 5961 Q11_42 + 5538 Q12_42 + 3761 Q13_42 + 3008 Q14_42 + 2944 Q15_42
   + 2871 Q16_42 + 2238 Q17_42 + 2083 Q18_42 + 1139 Q19_42 + 757 Q20_42
   + X2 <= 250000
98833 Q1_13 + 90526 Q2_13 + 61888 Q3_13 + 44370 Q4_13 + 23066 Q5_13
   + 18824 Q6_13 + 17037 Q7_13 + 11403 Q8_13 + 10387 Q9_13 + 10330 Q10_13
```

```
   + 5961 Q11_13 + 5538 Q12_13 + 3761 Q13_13 + 3008 Q14_13 + 2944 Q15_13
   + 2871 Q16_13 + 2238 Q17_13 + 2083 Q18_13 + 1139 Q19_13 + 757 Q20_13
   + X3 <= 250000
98833 Q1_23 + 90526 Q2_23 + 61888 Q3_23 + 44370 Q4_23 + 23066 Q5_23
   + 18824 Q6_23 + 17037 Q7_23 + 11403 Q8_23 + 10387 Q9_23 + 10330 Q10_23
   + 5961 Q11_23 + 5538 Q12_23 + 3761 Q13_23 + 3008 Q14_23 + 2944 Q15_23
   + 2871 Q16_23 + 2238 Q17_23 + 2083 Q18_23 + 1139 Q19_23 + 757 Q20_23
   + X3 <= 250000
98833 Q1_33 + 90526 Q2_33 + 61888 Q3_33 + 44370 Q4_33 + 23066 Q5_33
   + 18824 Q6_33 + 17037 Q7_33 + 11403 Q8_33 + 10387 Q9_33 + 10330 Q10_33
   + 5961 Q11_33 + 5538 Q12_33 + 3761 Q13_33 + 3008 Q14_33 + 2944 Q15_33
   + 2871 Q16_33 + 2238 Q17_33 + 2083 Q18_33 + 1139 Q19_33 + 757 Q20_33
   + X3 <= 250000
98833 Q1_43 + 90526 Q2_43 + 61888 Q3_43 + 44370 Q4_43 + 23066 Q5_43
   + 18824 Q6_43 + 17037 Q7_43 + 11403 Q8_43 + 10387 Q9_43 + 10330 Q10_43
   + 5961 Q11_43 + 5538 Q12_43 + 3761 Q13_43 + 3008 Q14_43 + 2944 Q15_43
   + 2871 Q16_43 + 2238 Q17_43 + 2083 Q18_43 + 1139 Q19_43 + 757 Q20_43
   + X3 <= 250000
98833 Q1_14 + 90526 Q2_14 + 61888 Q3_14 + 44370 Q4_14 + 23066 Q5_14
   + 18824 Q6_14 + 17037 Q7_14 + 11403 Q8_14 + 10387 Q9_14 + 10330 Q10_14
   + 5961 Q11_14 + 5538 Q12_14 + 3761 Q13_14 + 3008 Q14_14 + 2944 Q15_14
   + 2871 Q16_14 + 2238 Q17_14 + 2083 Q18_14 + 1139 Q19_14 + 757 Q20_14
   + X4 <= 250000
98833 Q1_24 + 90526 Q2_24 + 61888 Q3_24 + 44370 Q4_24 + 23066 Q5_24
   + 18824 Q6_24 + 17037 Q7_24 + 11403 Q8_24 + 10387 Q9_24 + 10330 Q10_24
   + 5961 Q11_24 + 5538 Q12_24 + 3761 Q13_24 + 3008 Q14_24 + 2944 Q15_24
   + 2871 Q16_24 + 2238 Q17_24 + 2083 Q18_24 + 1139 Q19_24 + 757 Q20_24
   + X4 <= 250000
98833 Q1_34 + 90526 Q2_34 + 61888 Q3_34 + 44370 Q4_34 + 23066 Q5_34
   + 18824 Q6_34 + 17037 Q7_34 + 11403 Q8_34 + 10387 Q9_34 + 10330 Q10_34
   + 5961 Q11_34 + 5538 Q12_34 + 3761 Q13_34 + 3008 Q14_34 + 2944 Q15_34
   + 2871 Q16_34 + 2238 Q17_34 + 2083 Q18_34 + 1139 Q19_34 + 757 Q20_34
   + X4 <= 250000
98833 Q1_44 + 90526 Q2_44 + 61888 Q3_44 + 44370 Q4_44 + 23066 Q5_44
   + 18824 Q6_44 + 17037 Q7_44 + 11403 Q8_44 + 10387 Q9_44 + 10330 Q10_44
   + 5961 Q11_44 + 5538 Q12_44 + 3761 Q13_44 + 3008 Q14_44 + 2944 Q15_44
   + 2871 Q16_44 + 2238 Q17_44 + 2083 Q18_44 + 1139 Q19_44 + 757 Q20_44
   + X4 <= 250000
157799 Q1_11 + 144536 Q2_11 + 98812 Q3_11 + 70842 Q4_11 + 36828 Q5_11
```

```
   + 30055 Q6_11 + 27202 Q7_11 + 18206 Q8_11 + 16584 Q9_11 + 16493 Q10_11
   + 9517 Q11_11 + 8842 Q12_11 + 6005 Q13_11 + 4803 Q14_11 + 4700 Q15_11
   + 4584 Q16_11 + 3573 Q17_11 + 3326 Q18_11 + 1819 Q19_11 + 1209 Q20_11
   <= 250000
157799 Q1_21 + 144536 Q2_21 + 98812 Q3_21 + 70842 Q4_21 + 36828 Q5_21
   + 30055 Q6_21 + 27202 Q7_21 + 18206 Q8_21 + 16584 Q9_21 + 16493 Q10_21
   + 9517 Q11_21 + 8842 Q12_21 + 6005 Q13_21 + 4803 Q14_21 + 4700 Q15_21
   + 4584 Q16_21 + 3573 Q17_21 + 3326 Q18_21 + 1819 Q19_21 + 1209 Q20_21
   <= 250000
157799 Q1_31 + 144536 Q2_31 + 98812 Q3_31 + 70842 Q4_31 + 36828 Q5_31
   + 30055 Q6_31 + 27202 Q7_31 + 18206 Q8_31 + 16584 Q9_31 + 16493 Q10_31
   + 9517 Q11_31 + 8842 Q12_31 + 6005 Q13_31 + 4803 Q14_31 + 4700 Q15_31
   + 4584 Q16_31 + 3573 Q17_31 + 3326 Q18_31 + 1819 Q19_31 + 1209 Q20_31
   <= 250000
157799 Q1_41 + 144536 Q2_41 + 98812 Q3_41 + 70842 Q4_41 + 36828 Q5_41
   + 30055 Q6_41 + 27202 Q7_41 + 18206 Q8_41 + 16584 Q9_41 + 16493 Q10_41
   + 9517 Q11_41 + 8842 Q12_41 + 6005 Q13_41 + 4803 Q14_41 + 4700 Q15_41
   + 4584 Q16_41 + 3573 Q17_41 + 3326 Q18_41 + 1819 Q19_41 + 1209 Q20_41
   <= 250000
157799 Q1_12 + 144536 Q2_12 + 98812 Q3_12 + 70842 Q4_12 + 36828 Q5_12
   + 30055 Q6_12 + 27202 Q7_12 + 18206 Q8_12 + 16584 Q9_12 + 16493 Q10_12
   + 9517 Q11_12 + 8842 Q12_12 + 6005 Q13_12 + 4803 Q14_12 + 4700 Q15_12
   + 4584 Q16_12 + 3573 Q17_12 + 3326 Q18_12 + 1819 Q19_12 + 1209 Q20_12
   <= 250000
157799 Q1_22 + 144536 Q2_22 + 98812 Q3_22 + 70842 Q4_22 + 36828 Q5_22
   + 30055 Q6_22 + 27202 Q7_22 + 18206 Q8_22 + 16584 Q9_22 + 16493 Q10_22
   + 9517 Q11_22 + 8842 Q12_22 + 6005 Q13_22 + 4803 Q14_22 + 4700 Q15_22
   + 4584 Q16_22 + 3573 Q17_22 + 3326 Q18_22 + 1819 Q19_22 + 1209 Q20_22
   <= 250000
157799 Q1_32 + 144536 Q2_32 + 98812 Q3_32 + 70842 Q4_32
   + 36828 Q5_32 + 30055 Q6_32 + 27202 Q7_32 + 18206 Q8_32 + 16584 Q9_32
   + 16493 Q10_32 + 9517 Q11_32 + 8842 Q12_32 + 6005 Q13_32 + 4803 Q14_32
   + 4700 Q15_32 + 4584 Q16_32 + 3573 Q17_32 + 3326 Q18_32 + 1819 Q19_32
   + 1209 Q20_32 <= 250000
157799 Q1_42 + 144536 Q2_42 + 98812 Q3_42 + 70842 Q4_42
   + 36828 Q5_42 + 30055 Q6_42 + 27202 Q7_42 + 18206 Q8_42 + 16584 Q9_42
   + 16493 Q10_42 + 9517 Q11_42 + 8842 Q12_42 + 6005 Q13_42 + 4803 Q14_42
   + 4700 Q15_42 + 4584 Q16_42 + 3573 Q17_42 + 3326 Q18_42 + 1819 Q19_42
   + 1209 Q20_42 <= 250000
```

```
 157799 Q1_13 + 144536 Q2_13 + 98812 Q3_13 + 70842 Q4_13
   + 36828 Q5_13 + 30055 Q6_13 + 27202 Q7_13 + 18206 Q8_13 + 16584 Q9_13
   + 16493 Q10_13 + 9517 Q11_13 + 8842 Q12_13 + 6005 Q13_13 + 4803 Q14_13
   + 4700 Q15_13 + 4584 Q16_13 + 3573 Q17_13 + 3326 Q18_13 + 1819 Q19_13
   + 1209 Q20_13 <= 250000
157799 Q1_23 + 144536 Q2_23 + 98812 Q3_23 + 70842 Q4_23
   + 36828 Q5_23 + 30055 Q6_23 + 27202 Q7_23 + 18206 Q8_23 + 16584 Q9_23
   + 16493 Q10_23 + 9517 Q11_23 + 8842 Q12_23 + 6005 Q13_23 + 4803 Q14_23
   + 4700 Q15_23 + 4584 Q16_23 + 3573 Q17_23 + 3326 Q18_23 + 1819 Q19_23
   + 1209 Q20_23 <= 250000
157799 Q1_33 + 144536 Q2_33 + 98812 Q3_33 + 70842 Q4_33
   + 36828 Q5_33 + 30055 Q6_33 + 27202 Q7_33 + 18206 Q8_33 + 16584 Q9_33
   + 16493 Q10_33 + 9517 Q11_33 + 8842 Q12_33 + 6005 Q13_33 + 4803 Q14_33
   + 4700 Q15_33 + 4584 Q16_33 + 3573 Q17_33 + 3326 Q18_33 + 1819 Q19_33
   + 1209 Q20_33 <= 250000
157799 Q1_43 + 144536 Q2_43 + 98812 Q3_43 + 70842 Q4_43
   + 36828 Q5_43 + 30055 Q6_43 + 27202 Q7_43 + 18206 Q8_43 + 16584 Q9_43
   + 16493 Q10_43 + 9517 Q11_43 + 8842 Q12_43 + 6005 Q13_43 + 4803 Q14_43
   + 4700 Q15_43 + 4584 Q16_43 + 3573 Q17_43 + 3326 Q18_43 + 1819 Q19_43
   + 1209 Q20_43 <= 250000
157799 Q1_14 + 144536 Q2_14 + 98812 Q3_14 + 70842 Q4_14
   + 36828 Q5_14 + 30055 Q6_14 + 27202 Q7_14 + 18206 Q8_14 + 16584 Q9_14
   + 16493 Q10_14 + 9517 Q11_14 + 8842 Q12_14 + 6005 Q13_14 + 4803 Q14_14
   + 4700 Q15_14 + 4584 Q16_14 + 3573 Q17_14 + 3326 Q18_14 + 1819 Q19_14
   + 1209 Q20_14 <= 250000
157799 Q1_24 + 144536 Q2_24 + 98812 Q3_24 + 70842 Q4_24
   + 36828 Q5_24 + 30055 Q6_24 + 27202 Q7_24 + 18206 Q8_24 + 16584 Q9_24
   + 16493 Q10_24 + 9517 Q11_24 + 8842 Q12_24 + 6005 Q13_24 + 4803 Q14_24
   + 4700 Q15_24 + 4584 Q16_24 + 3573 Q17_24 + 3326 Q18_24 + 1819 Q19_24
   + 1209 Q20_24 <= 250000
157799 Q1_34 + 144536 Q2_34 + 98812 Q3_34 + 70842 Q4_34
   + 36828 Q5_34 + 30055 Q6_34 + 27202 Q7_34 + 18206 Q8_34 + 16584 Q9_34
   + 16493 Q10_34 + 9517 Q11_34 + 8842 Q12_34 + 6005 Q13_34 + 4803 Q14_34
   + 4700 Q15_34 + 4584 Q16_34 + 3573 Q17_34 + 3326 Q18_34 + 1819 Q19_34
   + 1209 Q20_34 <= 250000
157799 Q1_44 + 144536 Q2_44 + 98812 Q3_44 + 70842 Q4_44
   + 36828 Q5_44 + 30055 Q6_44 + 27202 Q7_44 + 18206 Q8_44 + 16584 Q9_44
   + 16493 Q10_44 + 9517 Q11_44 + 8842 Q12_44 + 6005 Q13_44 + 4803 Q14_44
   + 4700 Q15_44 + 4584 Q16_44 + 3573 Q17_44 + 3326 Q18_44 + 1819 Q19_44
```

```
   + 1209 Q20_44 <= 250000
204813 Q21_11 + 203930 Q22_11 + 94158 Q23_11 + 71116 Q24_11
   + 70599 Q25_11 + 70393 Q26_11 + 55040 Q27_11 + 54002 Q28_11
   + 43197 Q29_11 + 41199 Q30_11 + 28041 Q31_11 + 18510 Q32_11
   + 18293 Q33_11 + 14854 Q34_11 + 12276 Q35_11 + 8152 Q36_11 + 7047 Q37_11
   + 6310 Q38_11 + 5033 Q39_11 + 1945 Q40_11 - X1 <= 0
204813 Q21_21 + 203930 Q22_21 + 94158 Q23_21 + 71116 Q24_21
   + 70599 Q25_21 + 70393 Q26_21 + 55040 Q27_21 + 54002 Q28_21
   + 43197 Q29_21 + 41199 Q30_21 + 28041 Q31_21 + 18510 Q32_21
   + 18293 Q33_21 + 14854 Q34_21 + 12276 Q35_21 + 8152 Q36_21 + 7047 Q37_21
   + 6310 Q38_21 + 5033 Q39_21 + 1945 Q40_21 - X1 <= 0
204813 Q21_31 + 203930 Q22_31 + 94158 Q23_31 + 71116 Q24_31
   + 70599 Q25_31 + 70393 Q26_31 + 55040 Q27_31 + 54002 Q28_31
   + 43197 Q29_31 + 41199 Q30_31 + 28041 Q31_31 + 18510 Q32_31
   + 18293 Q33_31 + 14854 Q34_31 + 12276 Q35_31 + 8152 Q36_31 + 7047 Q37_31
   + 6310 Q38_31 + 5033 Q39_31 + 1945 Q40_31 - X1 <= 0
204813 Q21_41 + 203930 Q22_41 + 94158 Q23_41 + 71116 Q24_41
   + 70599 Q25_41 + 70393 Q26_41 + 55040 Q27_41 + 54002 Q28_41
   + 43197 Q29_41 + 41199 Q30_41 + 28041 Q31_41 + 18510 Q32_41
   + 18293 Q33_41 + 14854 Q34_41 + 12276 Q35_41 + 8152 Q36_41 + 7047 Q37_41
   + 6310 Q38_41 + 5033 Q39_41 + 1945 Q40_41 - X1 <= 0
 204813 Q21_12 + 203930 Q22_12 + 94158 Q23_12 + 71116 Q24_12
   + 70599 Q25_12 + 70393 Q26_12 + 55040 Q27_12 + 54002 Q28_12
   + 43197 Q29_12 + 41199 Q30_12 + 28041 Q31_12 + 18510 Q32_12
   + 18293 Q33_12 + 14854 Q34_12 + 12276 Q35_12 + 8152 Q36_12 + 7047 Q37_12
   + 6310 Q38_12 + 5033 Q39_12 + 1945 Q40_12 - X2 <= 0
204813 Q21_22 + 203930 Q22_22 + 94158 Q23_22 + 71116 Q24_22
   + 70599 Q25_22 + 70393 Q26_22 + 55040 Q27_22 + 54002 Q28_22
   + 43197 Q29_22 + 41199 Q30_22 + 28041 Q31_22 + 18510 Q32_22
   + 18293 Q33_22 + 14854 Q34_22 + 12276 Q35_22 + 8152 Q36_22 + 7047 Q37_22
   + 6310 Q38_22 + 5033 Q39_22 + 1945 Q40_22 - X2 <= 0
204813 Q21_32 + 203930 Q22_32 + 94158 Q23_32 + 71116 Q24_32
   + 70599 Q25_32 + 70393 Q26_32 + 55040 Q27_32 + 54002 Q28_32
   + 43197 Q29_32 + 41199 Q30_32 + 28041 Q31_32 + 18510 Q32_32
   + 18293 Q33_32 + 14854 Q34_32 + 12276 Q35_32 + 8152 Q36_32 + 7047 Q37_32
   + 6310 Q38_32 + 5033 Q39_32 + 1945 Q40_32 - X2 <= 0
204813 Q21_42 + 203930 Q22_42 + 94158 Q23_42 + 71116 Q24_42
   + 70599 Q25_42 + 70393 Q26_42 + 55040 Q27_42 + 54002 Q28_42
   + 43197 Q29_42 + 41199 Q30_42 + 28041 Q31_42 + 18510 Q32_42
```

```
   + 18293 Q33_42 + 14854 Q34_42 + 12276 Q35_42 + 8152 Q36_42 + 7047 Q37_42
   + 6310 Q38_42 + 5033 Q39_42 + 1945 Q40_42 - X2 <= 0
204813 Q21_13 + 203930 Q22_13 + 94158 Q23_13 + 71116 Q24_13
   + 70599 Q25_13 + 70393 Q26_13 + 55040 Q27_13 + 54002 Q28_13
   + 43197 Q29_13 + 41199 Q30_13 + 28041 Q31_13 + 18510 Q32_13
   + 18293 Q33_13 + 14854 Q34_13 + 12276 Q35_13 + 8152 Q36_13 + 7047 Q37_13
   + 6310 Q38_13 + 5033 Q39_13 + 1945 Q40_13 - X3 <= 0
204813 Q21_23 + 203930 Q22_23 + 94158 Q23_23 + 71116 Q24_23
   + 70599 Q25_23 + 70393 Q26_23 + 55040 Q27_23 + 54002 Q28_23
   + 43197 Q29_23 + 41199 Q30_23 + 28041 Q31_23 + 18510 Q32_23
   + 18293 Q33_23 + 14854 Q34_23 + 12276 Q35_23 + 8152 Q36_23 + 7047 Q37_23
   + 6310 Q38_23 + 5033 Q39_23 + 1945 Q40_23 - X3 <= 0
204813 Q21_33 + 203930 Q22_33 + 94158 Q23_33 + 71116 Q24_33
   + 70599 Q25_33 + 70393 Q26_33 + 55040 Q27_33 + 54002 Q28_33
   + 43197 Q29_33 + 41199 Q30_33 + 28041 Q31_33 + 18510 Q32_33
   + 18293 Q33_33 + 14854 Q34_33 + 12276 Q35_33 + 8152 Q36_33 + 7047 Q37_33
   + 6310 Q38_33 + 5033 Q39_33 + 1945 Q40_33 - X3 <= 0
204813 Q21_43 + 203930 Q22_43 + 94158 Q23_43 + 71116 Q24_43
   + 70599 Q25_43 + 70393 Q26_43 + 55040 Q27_43 + 54002 Q28_43
   + 43197 Q29_43 + 41199 Q30_43 + 28041 Q31_43 + 18510 Q32_43
   + 18293 Q33_43 + 14854 Q34_43 + 12276 Q35_43 + 8152 Q36_43 + 7047 Q37_43
   + 6310 Q38_43 + 5033 Q39_43 + 1945 Q40_43 - X3 <= 0
204813 Q21_14 + 203930 Q22_14 + 94158 Q23_14 + 71116 Q24_14
   + 70599 Q25_14 + 70393 Q26_14 + 55040 Q27_14 + 54002 Q28_14
   + 43197 Q29_14 + 41199 Q30_14 + 28041 Q31_14 + 18510 Q32_14
   + 18293 Q33_14 + 14854 Q34_14 + 12276 Q35_14 + 8152 Q36_14 + 7047 Q37_14
   + 6310 Q38_14 + 5033 Q39_14 + 1945 Q40_14 - X4 <= 0
204813 Q21_24 + 203930 Q22_24 + 94158 Q23_24 + 71116 Q24_24
   + 70599 Q25_24 + 70393 Q26_24 + 55040 Q27_24 + 54002 Q28_24
   + 43197 Q29_24 + 41199 Q30_24 + 28041 Q31_24 + 18510 Q32_24
   + 18293 Q33_24 + 14854 Q34_24 + 12276 Q35_24 + 8152 Q36_24 + 7047 Q37_24
   + 6310 Q38_24 + 5033 Q39_24 + 1945 Q40_24 - X4 <= 0
204813 Q21_34 + 203930 Q22_34 + 94158 Q23_34 + 71116 Q24_34
   + 70599 Q25_34 + 70393 Q26_34 + 55040 Q27_34 + 54002 Q28_34
   + 43197 Q29_34 + 41199 Q30_34 + 28041 Q31_34 + 18510 Q32_34
   + 18293 Q33_34 + 14854 Q34_34 + 12276 Q35_34 + 8152 Q36_34 + 7047 Q37_34
   + 6310 Q38_34 + 5033 Q39_34 + 1945 Q40_34 - X4 <= 0
204813 Q21_44 + 203930 Q22_44 + 94158 Q23_44 + 71116 Q24_44
   + 70599 Q25_44 + 70393 Q26_44 + 55040 Q27_44 + 54002 Q28_44
```

```
   + 43197 Q29_44 + 41199 Q30_44 + 28041 Q31_44 + 18510 Q32_44
   + 18293 Q33_44 + 14854 Q34_44 + 12276 Q35_44 + 8152 Q36_44 + 7047 Q37_44
   + 6310 Q38_44 + 5033 Q39_44 + 1945 Q40_44 - X4 <= 0
Bounds
 X1 <= 250000
 X2 <= 250000
 X3 <= 250000
 X4 <= 250000
 Binaries
 Q1_11 Q1_21 Q1_31 Q1_41 Q1_12 Q1_22 Q1_32 Q1_42 Q1_13 Q1_23 Q1_33 Q1_43
 Q1_14 Q1_24 Q1_34 Q1_44 Q2_11 Q2_21 Q2_31 Q2_41 Q2_12 Q2_22 Q2_32 Q2_42
 Q2_13 Q2_23 Q2_33 Q2_43 Q2_14 Q2_24 Q2_34 Q2_44 Q3_11 Q3_21 Q3_31 Q3_41
 Q3_12 Q3_22 Q3_32 Q3_42 Q3_13 Q3_23 Q3_33 Q3_43 Q3_14 Q3_24 Q3_34 Q3_44
 Q4_11 Q4_21 Q4_31 Q4_41 Q4_12 Q4_22 Q4_32 Q4_42 Q4_13 Q4_23 Q4_33 Q4_43
 Q4_14 Q4_24 Q4_34 Q4_44 Q5_11 Q5_21 Q5_31 Q5_41 Q5_12 Q5_22 Q5_32 Q5_42
 Q5_13 Q5_23 Q5_33 Q5_43 Q5_14 Q5_24 Q5_34 Q5_44 Q6_11 Q6_21 Q6_31 Q6_41
 Q6_12 Q6_22 Q6_32 Q6_42 Q6_13 Q6_23 Q6_33 Q6_43 Q6_14 Q6_24 Q6_34 Q6_44
 Q7_11 Q7_21 Q7_31 Q7_41 Q7_12 Q7_22 Q7_32 Q7_42 Q7_13 Q7_23 Q7_33 Q7_43
 Q7_14 Q7_24 Q7_34 Q7_44 Q8_11 Q8_21 Q8_31 Q8_41 Q8_12 Q8_22 Q8_32 Q8_42
 Q8_13 Q8_23 Q8_33 Q8_43 Q8_14 Q8_24 Q8_34 Q8_44 Q9_11 Q9_21 Q9_31 Q9_41
 Q9_12 Q9_22 Q9_32 Q9_42 Q9_13 Q9_23 Q9_33 Q9_43 Q9_14 Q9_24 Q9_34 Q9_44
 Q10_11 Q10_21 Q10_31 Q10_41 Q10_12 Q10_22 Q10_32 Q10_42 Q10_13 Q10_23
 Q10_33 Q10_43 Q10_14 Q10_24 Q10_34 Q10_44 Q11_11 Q11_21 Q11_31 Q11_41
 Q11_12 Q11_22 Q11_32 Q11_42 Q11_13 Q11_23 Q11_33 Q11_43 Q11_14 Q11_24
 Q11_34 Q11_44 Q12_11 Q12_21 Q12_31 Q12_41 Q12_12 Q12_22 Q12_32 Q12_42
 Q12_13 Q12_23 Q12_33 Q12_43 Q12_14 Q12_24 Q12_34 Q12_44 Q13_11 Q13_21
 Q13_31 Q13_41 Q13_12 Q13_22 Q13_32 Q13_42 Q13_13 Q13_23 Q13_33 Q13_43
 Q13_14 Q13_24 Q13_34 Q13_44 Q14_11 Q14_21 Q14_31 Q14_41 Q14_12 Q14_22
 Q14_32 Q14_42 Q14_13 Q14_23 Q14_33 Q14_43 Q14_14 Q14_24 Q14_34 Q14_44
 Q15_11 Q15_21 Q15_31 Q15_41 Q15_12 Q15_22 Q15_32 Q15_42 Q15_13 Q15_23
 Q15_33 Q15_43 Q15_14 Q15_24 Q15_34 Q15_44 Q16_11 Q16_21 Q16_31 Q16_41
 Q16_12 Q16_22 Q16_32 Q16_42 Q16_13 Q16_23 Q16_33 Q16_43 Q16_14 Q16_24
 Q16_34 Q16_44 Q17_11 Q17_21 Q17_31 Q17_41 Q17_12 Q17_22 Q17_32 Q17_42
 Q17_13 Q17_23 Q17_33 Q17_43 Q17_14 Q17_24 Q17_34 Q17_44 Q18_11 Q18_21
 Q18_31 Q18_41 Q18_12 Q18_22 Q18_32 Q18_42 Q18_13 Q18_23 Q18_33 Q18_43
 Q18_14 Q18_24 Q18_34 Q18_44 Q19_11 Q19_21 Q19_31 Q19_41 Q19_12 Q19_22
 Q19_32 Q19_42 Q19_13 Q19_23 Q19_33 Q19_43 Q19_14 Q19_24 Q19_34 Q19_44
 Q20_11 Q20_21 Q20_31 Q20_41 Q20_12 Q20_22 Q20_32 Q20_42 Q20_13 Q20_23
 Q20_33 Q20_43 Q20_14 Q20_24 Q20_34 Q20_44 Q21_11 Q21_21 Q21_31 Q21_41
```

Q21_12 Q21_22 Q21_32 Q21_42 Q21_13 Q21_23 Q21_33 Q21_43 Q21_14 Q21_24
Q21_34 Q21_44 Q22_11 Q22_21 Q22_31 Q22_41 Q22_12 Q22_22 Q22_32 Q22_42
Q22_13 Q22_23 Q22_33 Q22_43 Q22_14 Q22_24 Q22_34 Q22_44 Q23_11 Q23_21
Q23_31 Q23_41 Q23_12 Q23_22 Q23_32 Q23_42 Q23_13 Q23_23 Q23_33 Q23_43
Q23_14 Q23_24 Q23_34 Q23_44 Q24_11 Q24_21 Q24_31 Q24_41 Q24_12 Q24_22
Q24_32 Q24_42 Q24_13 Q24_23 Q24_33 Q24_43 Q24_14 Q24_24 Q24_34 Q24_44
Q25_11 Q25_21 Q25_31 Q25_41 Q25_12 Q25_22 Q25_32 Q25_42 Q25_13 Q25_23
Q25_33 Q25_43 Q25_14 Q25_24 Q25_34 Q25_44 Q26_11 Q26_21 Q26_31 Q26_41
Q26_12 Q26_22 Q26_32 Q26_42 Q26_13 Q26_23 Q26_33 Q26_43 Q26_14 Q26_24
Q26_34 Q26_44 Q27_11 Q27_21 Q27_31 Q27_41 Q27_12 Q27_22 Q27_32 Q27_42
Q27_13 Q27_23 Q27_33 Q27_43 Q27_14 Q27_24 Q27_34 Q27_44 Q28_11 Q28_21
Q28_31 Q28_41 Q28_12 Q28_22 Q28_32 Q28_42 Q28_13 Q28_23 Q28_33 Q28_43
Q28_14 Q28_24 Q28_34 Q28_44 Q29_11 Q29_21 Q29_31 Q29_41 Q29_12 Q29_22
Q29_32 Q29_42 Q29_13 Q29_23 Q29_33 Q29_43 Q29_14 Q29_24 Q29_34 Q29_44
Q30_11 Q30_21 Q30_31 Q30_41 Q30_12 Q30_22 Q30_32 Q30_42 Q30_13 Q30_23
Q30_33 Q30_43 Q30_14 Q30_24 Q30_34 Q30_44 Q31_11 Q31_21 Q31_31 Q31_41
Q31_12 Q31_22 Q31_32 Q31_42 Q31_13 Q31_23 Q31_33 Q31_43 Q31_14 Q31_24
Q31_34 Q31_44 Q32_11 Q32_21 Q32_31 Q32_41 Q32_12 Q32_22 Q32_32 Q32_42
Q32_13 Q32_23 Q32_33 Q32_43 Q32_14 Q32_24 Q32_34 Q32_44 Q33_11 Q33_21
Q33_31 Q33_41 Q33_12 Q33_22 Q33_32 Q33_42 Q33_13 Q33_23 Q33_33 Q33_43
Q33_14 Q33_24 Q33_34 Q33_44 Q34_11 Q34_21 Q34_31 Q34_41 Q34_12 Q34_22
Q34_32 Q34_42 Q34_13 Q34_23 Q34_33 Q34_43 Q34_14 Q34_24 Q34_34 Q34_44
Q35_11 Q35_21 Q35_31 Q35_41 Q35_12 Q35_22 Q35_32 Q35_42 Q35_13 Q35_23
Q35_33 Q35_43 Q35_14 Q35_24 Q35_34 Q35_44 Q36_11 Q36_21 Q36_31 Q36_41
Q36_12 Q36_22 Q36_32 Q36_42 Q36_13 Q36_23 Q36_33 Q36_43 Q36_14 Q36_24
Q36_34 Q36_44 Q37_11 Q37_21 Q37_31 Q37_41 Q37_12 Q37_22 Q37_32 Q37_42
Q37_13 Q37_23 Q37_33 Q37_43 Q37_14 Q37_24 Q37_34 Q37_44 Q38_11 Q38_21
Q38_31 Q38_41 Q38_12 Q38_22 Q38_32 Q38_42 Q38_13 Q38_23 Q38_33 Q38_43
Q38_14 Q38_24 Q38_34 Q38_44 Q39_11 Q39_21 Q39_31 Q39_41 Q39_12 Q39_22
Q39_32 Q39_42 Q39_13 Q39_23 Q39_33 Q39_43 Q39_14 Q39_24 Q39_34 Q39_44
Q40_11 Q40_21 Q40_31 Q40_41 Q40_12 Q40_22 Q40_32 Q40_42 Q40_13 Q40_23
Q40_33 Q40_43 Q40_14 Q40_24 Q40_34 Q40_44

Generals

X1 X2 X3 X4

End

Bibliography

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 4–13, 1998.

[2] Z. Al-bayati, B. H. Meyer, and H. Zeng. Fault-tolerant scheduling of multicore mixed-criticality systems under permanent failures. In *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 57–62, Sept 2016.

[3] J. H. Anderson, S. K. Baruah, and B. B. Brandenburg. Multicore operating-system support for mixed criticality. Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification, San Francisco, 2009.

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

[5] N. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline-monotonic approach. In *in Proc. IEEE Workshop on Real-Time Operating Systems and Software*, pages 133–137, 1991.

[6] N. C. Audsley. On priority asignment in fixed priority scheduling. *Inf. Process. Lett.*, 79(1):39–44, May 2001.

[7] H. Aysan, R. Dobrin, and S. Punnekkat. Fault tolerant scheduling on controller area network (can). In *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, pages 226–232, May 2010.

[8] T. Baker and A. Shaw. The cyclic executive model and ada. In *Real-Time Systems Symposium, 1988., Proceedings.*, pages 120–129, Dec 1988.

[9] S. Baruah. Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors. *Computers, IEEE Transactions on*, 53(6):781–784, 2004.

[10] S. Baruah. Certification-cognizant scheduling of tasks with pessimistic frequency specification. In test, editor, *Industrial Embedded Systems (SIES), 2012 7th IEEE International Symposium on*, pages 31 –38, June 2012.

[11] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 145 –154, July 2012.

[12] S. Baruah, V. Bonifaci, G. D'angelo, H. Li, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie. Preemptive uniprocessor scheduling of mixed-criticality sporadic task systems. *J. ACM*, 62(2):14:1–14:33, May 2015.

[13] S. Baruah, V. Bonifaci, G. D'Angelo, A. Marchetti-Spaccamela, S. Ster, and L. Stougie. Mixed-criticality scheduling of sporadic task systems. In C. Demetrescu and M. HalldÃşrsson, editors, *Algorithms âĂŞ ESA 2011*, volume 6942 of *Lecture Notes in Computer Science*, pages 555–566. Springer Berlin Heidelberg, 2011.

[14] S. Baruah and A. Burns. Implementing mixed criticality systems in Ada. In A. Romanovsky and T. Vardanega, editors, *Reliable Software Technologies - Ada-Europe 2011*, volume 6652 of *Lecture Notes in Computer Science*, pages 174–188. Springer Berlin Heidelberg, 2011.

[15] S. Baruah, A. Burns, and R. Davis. Response-time analysis for mixed criticality systems. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 34 –43, 29 2011-Dec. 2 2011.

[16] S. Baruah and B. Chattopadhyay. Response-time analysis of mixed criticality systems with pessimistic frequency specification. Technical report, University of North Carolina at Chapel Hill, 2013.

[17] S. Baruah, B. Chattopadhyay, H. Li, and I. Shin. Mixed-criticality scheduling on multiprocessors. *Real-Time Systems*, pages 1–36, 2013.

[18] S. Baruah and G. Fohler. Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 3 –12, 29 2011-Dec. 2 2011.

[19] S. Baruah and Z. Guo. Mixed-criticality scheduling upon varying-speed processors. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, pages 68–77, Dec 2013.

[20] S. Baruah, H. Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, pages 13 –22, April 2010.

[21] S. Baruah and S. Vestal. Schedulability analysis of sporadic tasks with multiple criticality specifications. In *Real-Time Systems, 2008. ECRTS '08. Euromicro Conference on*, pages 147 –155, July 2008.

[22] A. Bastoni, B. Brandenburg, and J. Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. *Proceedings of OSPERT*, pages 33–44, 2010.

[23] A. Bastoni, B. Brandenburg, and J. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 14–24, 2010.

[24] M. Berkelaar, K. Eikland, and P. Notebaert. lp_solve 5.5, open source (mixed-integer) linear programming system. Software, May 1 2004. Available at <http://lpsolve.sourceforge.net/5.5/>.

[25] E. Bini and G. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[26] B. B. Brandenburg. *Scheduling and locking in multiprocessor real-time operating systems*. PhD thesis, The University of North Carolina, Chapel Hill, 2011.

[27] A. Burns. An augmented model for mixed criticality. In D. Baruah, Cucu-Grosjean and Maiza, editors, *Mixed Criticality on Multicore/Manycore Platforms (Dagstuhl Seminar 15121)*, volume 5, pages 92–93. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2015.

[28] A. Burns and S. Baruah. Timing faults and mixed criticality systems. In C. Jones and J. Lloyd, editors, *Dependable and Historic Computing*, volume 6875 of *Lecture Notes in Computer Science*, pages 147–166. Springer Berlin Heidelberg, 2011.

[29] A. Burns and S. Baruah. Semi-partitioned cyclic executives for mixed criticality systems. volume WMC RTSS 2015, pages 7–12, 2015.

[30] A. Burns and R. Davis. Mixed criticality on controller area network. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 125–134, 2013.

[31] M. Chisholm, B. C. Ward, N. Kim, and J. H. Anderson. Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In *Real-Time Systems Symposium, 2015 IEEE*, pages 305–316, Dec 2015.

[32] G. Dantzig. *Linear Programming and Extensions*. Landmarks in Physics and Mathematics. Princeton University Press, 1963.

[33] S. K. Dhall and C. L. Iiu. On a real-time scheduling problem. *Operations Research*, 26(1):127, 1978.

[34] A. Easwaran. Demand-based scheduling of mixed-criticality sporadic tasks on one processor. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, pages 78–87, Dec 2013.

[35] P. Ekberg and W. Yi. Bounding and shaping the demand of mixed-criticality sporadic tasks. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 135–144, 2012.

[36] P. Ekberg and W. Yi. Bounding and shaping the demand of generalized mixed-criticality sporadic task systems. *Real-Time Systems*, 50(1):48–86, 2014.

[37] T. Fleming. Extending mixed criticality scheduling. Master's thesis, University Of York, 04 2013.

[38] T. Fleming and A. Burns. Extending mixed criticality scheduling. volume WMC RTSS 2013, pages 7–12, 2013.

[39] T. Fleming, H.-M. Huang, A. Burns, C. Gill, S. Baruah, and C. Lu. Corrections to and discussion of "implementation and evaluation of mixed-criticality scheduling approaches for sporadic tasks". *ACM Trans. Embed. Comput. Syst.*, 16(77), April 2017.

[40] M. R. Garey and D. S. Johnson. *Computers and intractability*, volume 29. WH Freeman New York, 2002.

[41] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *Embedded Software (EMSOFT), 2013 Proceedings of the International Conference on*, pages 1–15, Sept 2013.

[42] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele. Mapping mixed-criticality applications on multi-core architectures. In *Design, Automation & Test in Europe Conference (DATE), Hot-Topic Session on Predictable Multi-core Computing*, Dresden, Germany, Mar 2014. IEEE.

[43] F. Glover. Tabu search-part i. *ORSA Journal on computing*, 1(3):190–206, 1989.

[44] D. Goswami, M. Lukasiewycz, R. Schneider, and S. Chakraborty. Time-triggered implementations of mixed-criticality automotive software. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, pages 1227–1232, San Jose, CA, USA, 2012. EDA Consortium.

[45] I. Gurobi Optimization. Gurobi optimizer 7.0. http://www.gurobi.com/.

[46] J. L. Herman, C. J. Kenna, M. S. Mollison, J. H. Anderson, and D. M. Johnson. RTOS support for multicore mixed-criticality systems. In *Proceedings of the 2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, RTAS '12, pages 197–208, Washington, DC, USA, 2012. IEEE Computer Society.

[47] H.-M. Huang, C. Gill, and C. Lu. Implementation and evaluation of mixed-criticality scheduling approaches for sporadic tasks. *ACM Trans. Embed. Comput. Syst.*, 13(4s):126:1–126:25, Apr. 2014.

[48] P. Huang, G. Giannopoulou, N. Stoimenov, and L. Thiele. Service adaptions for mixed-criticality systems. In *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*, pages 125–130, Jan 2014.

[49] IBM. Cplex optimizer. https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/.

[50] M. Jan, L. Zaourar, V. Legout, and L. Pautet. Handling criticality mode change in time-triggered systems through linear programming. In *Ada User Journal, Proc of Workshop on Mixed Criticality for Industrial Systems (WMCISâĂŹ2014)*, volume 35, pages 138–143, 2014.

[51] M. Jan, L. Zaourar, and M. Pitel. Maximizing the execution rate of low-criticality tasks in mixed criticality systems. volume WMC RTSS 2013, pages 7–12, 2013.

[52] L. V. Kantorovich. Mathematical methods of organizing and planning production. *Management Science*, 6(4):366–422, 1960.

[53] O. Kelly, H. Aydin, and B. Zhao. On partitioned scheduling of fixed-priority mixed-criticality task sets. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pages 1051 –1059, nov. 2011.

[54] R. Kirner, S. Iacovelli, and M. Zolda. Optimised adaptation of mixed-criticality systems with periodic tasks on uniform multiprocessors in case of faults. In

2015 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops, pages 17–25, April 2015.

[55] K. Lakshmanan, D. de Niz, R. Rajkumar, and G. Moreno. Resource allocation in distributed mixed-criticality cyber-physical systems. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 169 –178, June 2010.

[56] V. Legout, M. Jan, and L. Pautet. Mixed-criticality multiprocessor real-time systems: Energy consumption vs deadline misses. In *First Workshop on Real-Time Mixed Criticality Systems (ReTiMiCS)*, pages 1–6, Taipei, Taiwan, Aug. 2013.

[57] V. Legout, M. Jan, and L. Pautet. A scheduling algorithm to reduce the static energy consumption of multiprocessor real-time systems. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, RTNS '13, pages 99–108, New York, NY, USA, 2013. ACM.

[58] V. Legout, M. Jan, and L. Pautet. Scheduling algorithms to reduce the static energy consumption of real-time systems. *Real-Time Systems*, 51(2):153–191, 2015.

[59] H. Li and S. Baruah. Load-based schedulability analysis of certifiable mixed-criticality systems. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '10, pages 99–108, New York, NY, USA, 2010. ACM.

[60] H. Li and S. Baruah. Global mixed-criticality scheduling on multiprocessors. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 166 –175, July 2012.

[61] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, Jan. 1973.

[62] MATLAB. *version 9.2.0 (R2017a)*. The MathWorks Inc., Natick, Massachusetts, 2017.

[63] M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, J. A. Scoredos, and N. G. Corporation. Mixed-criticality real-time scheduling for multicore systems, 2010.

[64] D. Niz, K. Lakshmanan, and R. Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 291 –300, dec. 2009.

[65] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement. Technical Report 2013-10, Informatik, 2013.

[66] T. Park and S. Kim. Dynamic scheduling algorithm and its schedulability analysis for certifiable dual-criticality systems. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, pages 253 –262, October 2011.

[67] R. Pathan. Schedulability analysis of mixed-criticality systems on multiprocessors. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 309 –320, July 2012.

[68] D. P. Rodgers. Improvements in multiprocessor system design. *SIGARCH Comput. Archit. News*, 13(3):225–231, June 1985.

[69] P. Saraswat, P. Pop, and J. Madsen. Task mapping and bandwidth reservation for mixed hard/soft fault-tolerant embedded systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, pages 89–98, April.

[70] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga. Time-triggered mixed-critical scheduler. volume WMC RTSS, pages 67 – 72, 2013.

[71] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga. Time-triggered mixed-critical scheduler on single and multi-processor platforms. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Se-*

curity, and 2015 IEEE 12th International Conference on Embedded Software and Systems, pages 684–687, Aug 2015.

[72] D. Tamas-Selicean and P. Pop. Optimization of time-partitions for mixed-criticality real-time distributed embedded systems. In *Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2011 14th IEEE International Symposium on*, pages 1 –10, March 2011.

[73] D. Tamas-Selicean and P. Pop. Task mapping and partition allocation for mixed-criticality real-time systems. In *Dependable Computing (PRDC), 2011 IEEE 17th Pacific Rim International Symposium on*, pages 282 –283, December 2011.

[74] J. Theis and G. Fohler. Mixed criticality scheduling in time-triggered legacy systems. In *1st Workshop on Mixed Criticality Systems, IEEE Real-Time Systems Symposium*, December 2013.

[75] J. Theis, G. Fohler, and S. Baruah. Schedule table generation for time-triggered mixed criticality systems. In *1st Workshop on Mixed Criticality Systems, IEEE Real-Time Systems Symposium*, December 2013.

[76] A. Thekkilakattil, R. Dobrin, and S. Punnekkat. Mixed criticality scheduling in fault-tolerant distributed real-time systems. In *Embedded Systems (ICES), 2014 International Conference on*, pages 92–97. IEEE, 2014.

[77] M. J. Todd. The many facets of linear programming. *Mathematical Programming*, 91(3):417–436, 2002.

[78] S. Trujillo, A. Crespo, A. Alonso, and J. PÃľrez. Multipartes: Multi-core partitioning and virtualization for easing the certification of mixed-criticality systems. *Microprocessors and Microsystems*, 38(8, Part B):921 – 932, 2014.

[79] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.

[80] R. Vanderbei. *Linear Programming: Foundations and Extensions*. International Series in Operations Research & Management Science. Springer US, 2013.

[81] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pages 239 –243, December 2007.

[82] Y. Xie, L. Liu, R. Li, J. Hu, Y. Han, and X. Peng. Security-aware signal packing algorithm for can-based automotive cyber-physical systems. *IEEE/CAA Journal of Automatica Sinica*, 2(4):422–430, October 2015.

[83] Q. Zhao, Z. Gu, and H. Zeng. Pt-amc: Integrating preemption thresholds into mixed-criticality scheduling. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 141–146, 2013.