# Advancing Mixed Criticality Scheduling Techniques to Support Industrial Applications

**Stephen Andrew Law**

Doctor of Philosophy

Computer Science

University of York

May 2020

# Abstract

Safety critical software development is an extremely costly endeavour; software developers must forever target efficient processes that reduce software cost, while allowing significant increases in system size. The key challenge being how to reduce software cost, without compromising safety or quality.

The focus of this thesis is to research the development and temporal proof of a mixed criticality system. The thesis, which attempts to define an end to end process, begins by studying appropriate and efficient methods for assessing the timing performance of system components. The key being an approach that can be applied automatically at an early point in the design lifecycle.

The thesis then progresses to study how existing mixed criticality research needs to be advanced and matured in order to support an industrial safety critical application. This includes the definition of a scheduling model designed to provide the necessary protections advised by international aviation guidelines. In the final part of this thesis the timing process and mixed criticality system model are brought together to explore how a real system using these techniques could be validated.

# Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this or any other University. All sources are acknowledged as references. Parts of this thesis have been published in or submitted to:

S. Law, and I. Bate, *Achieving appropriate test coverage for reliable measurement-based timing analysis*, Proceedings of the 28th Euromicro Conference on Real-Time Systems (ECRTS), 2016.

S. Law, I. Bate, and B. Lesage, *Industrial Application of a Partitioning Scheduler to Support Mixed Criticality Systems*, Proceedings of the 31st Euromicro Conference on Real-Time Systems (ECRTS), 2019.

S. Law, B. Lesage, and I. Bate , *Justifying the Service Provided to Low-Criticality Tasks in a Mixed-Criticality System*, Proceedings of the 28th International Conference on Real Time Networks and Systems (RTNS), 2020.

Additionally the research undertaken in Chapter 3 has been extended by a fellow researcher and is published in the following paper:

B. Lesage, S. Law, and I. Bate, *TACO: An industrial case study of Test Automation for COverage*, Proceedings of the 26th International Conference on Real-Time Networks and Systems (RTNS), 2018.

The paper shows that the approach described in Chapter 3 can be scaled to analyse a full industrial system. The results of this analysis,

which were obtained from applying the approach to a live Rolls-Royce aircraft engine control project, are being used to support certification of the project.

# Contents

# List of Figures

# List of Tables

# List of Symbols

| Symbol | Meaning |
|---|---|
| $T_i$ | The period of task $i$ |
| $L_i$ | The criticality of task $i$ |
| $C_i^L$ | The worst case execution time of task $i$, which may optionally have a criticality $L$ |
| $D_i$ | The deadline of task $i$ |
| $R_i$ | The worst case response time of task $i$ |
| $J_i$ | The completion jitter time influencing task $i$ |
| $P_i$ | The priority of task $i$ |
| $S_i$ | The number of consecutive job skips that a robust task $i$ can support before failing to comply with the task's temporal requirements |
| $JF$ | The number of job failures measured within a system |
| $F$ | The number of job failures a system can tolerate without jobs being dropped or deadlines missed |
| $M$ | The number of job failures a system can tolerate without deadlines being missed, once all robust tasks have dropped $S_i$ jobs |

# Acknowledgement

I am grateful to my colleagues at Rolls-Royce for their encouragement and guidance on my work, and for providing the time I needed to complete this thesis. In particular, I would like to thank Ivan Ellis, Phil Elliot, Jeff Hobday and Guy Partridge. Furthermore, for their time and patience spent reviewing my work: Duncan Brown and Stuart Hutchesson. Thanks also go to colleagues at the University of York for their fruitful discussions, including Benjamin Lesage.

Special thanks go to my supervisors, and friends, at Rolls-Royce and York: Mike Bennett and Iain Bate. Thanks for your advice and guidance, but in particular thanks for picking me up during the times over the last (almost) seven years it felt as if finishing this work was not possible.

Most of all though, thanks to Kat. You never fail to support and believe in me. Thanks for your patience during the weekends and evenings you have spent talking to the back of my laptop screen, and especially thanks for spending your well-timed maternity leave proof reading this thesis[1]. I owe you more than I can say.

[1] and raising 678 comments, mostly about, poorly placed, commas - I hope I caught them all,.

# Chapter 1

# Introduction

One of the greatest problems facing safety critical software developers is that of software cost [1]. This is a problem that has plagued the software development industry for a number of years. However, the ever greater reliance on software controlled systems, coupled with the need for constantly increasing software features means this ever present problem must be continually addressed.

The ever advancing march towards increased efficiency and feature base ultimately guides software systems down a path of ever greater complexity and ever finer control. However, this increase in software scale can only be achieved if software costs on a per line basis are reduced. For instance, in the last thirty years the amount of software on board Boeing aircraft has roughly doubled every two years [2][3]. For a commercial company to remain competitive it cannot simply be assumed that the total cost of the software system will increase at the same rate.

Therefore, the desire for ever more complex system features must be met by equivalent reductions in software development costs. The problem is exemplified by the fact that safety criticality software is relied upon to perform in accordance with its requirements. Therefore, it is es-

sential that corners are not cut and that software quality is not adversely affected. Indeed, cost reduction must not be allowed to impact on the safety of the developed system [4], [5].

One of the principal reasons that safety critical components are so expensive to develop, is that they must be developed against robust processes designed to provide safety assurance against all use cases. Such components can be found in a multitude of industries and products; such as Defence, Nuclear, Marine, Rail and Avionics, amongst others. Typically, development of such components follows the guidance or standards set out in one of the many global guideline documents such as ISO26262, EN50128 and DO-178C; used for the automotive, rail and aerospace industries respectively. There are a significant number of parallels across each standard, particularly in the use of 'Integrity' or 'Assurance' levels, which represent a classification used to define each software component based on its impact to wider system safety should said component fail to execute in accordance with its requirements. In this thesis, the term is considered synonymous with a component's 'criticality'.

The focus of this thesis is on the development of avionics systems, according to DO-178C [6]. Therefore the term used to describe a software component's criticality is the Development Assurance Level, or DAL, as originally defined by ARP4754 [7] and used throughout DO-178C.

The software aspects of the aircraft and engine certification process assures that the software systems and components are ready for deployment in service. DO-178C contains a comprehensive set of objectives that should be fulfilled to certify each component, thus providing confidence in its operation. The greater the DAL (the higher the DAL) assigned to each software component, the greater the level of confidence should be achieved, and therefore the more certification objectives that need to be

accomplished. In other words the greater the consequence of failure of the software component, the greater the effort required to develop and assure said component.

Ensuring conformance to high DAL software certification objectives can be a laborious and expensive process. It involves confirming that each high level requirement traces through architecture and design to the code that implements it, and vice versa. The software must then be reviewed and analysed to confirm conformance to standard as well as confirming the accuracy and consistency of the software. Tests should be derived according to each requirement, which in turn should be shown to test the entire code base. The rigour and the process followed should be dictated by the DAL of the system.

The methods employed to facilitate the reduction of software cost in an industrial scale project is a topic large enough to fill several theses. Instead, in this thesis, as set out in the remainder of the introduction, software cost reduction is targeted by focusing on real time system aspects of software development. Firstly, appropriately automated and efficient methods of gaining confidence in a software component's temporal operation, its Worst Case Execution Time (WCET), are researched. Secondly, this thesis studies how these WCET results can be used along with new processes, techniques and architectures, to allow components of different criticalities to co-exist on the same processing platform; thus allowing system developers to better target their certification effort. The research undertaken identified a number of significant issues that needed to be addressed with the existing published literature in academia, this thesis provides extensions to address said shortfalls.

This introduction now steps through some of the issues that this thesis aims to address.

## 1.1 Software Development Life-cycle

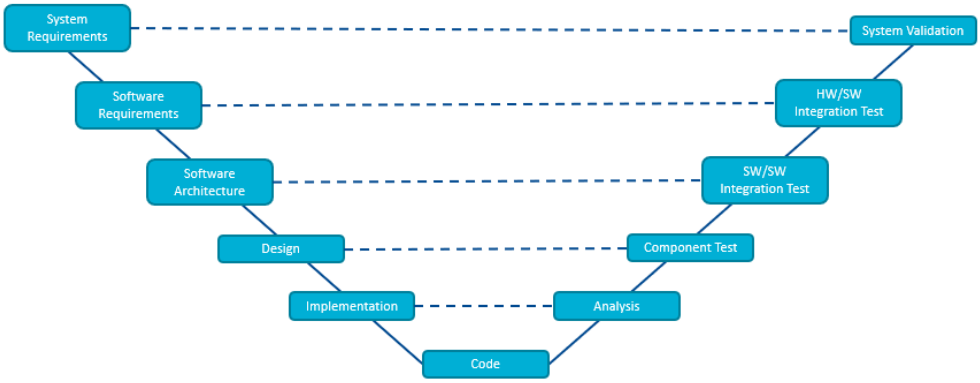


Figure 1.1: Typical Software Development 'V' lifecycle.

A typical software design lifecycle, known as the V-model, is illustrated in Figure 1.1. The lifecycle follows a process on the left hand side of progressively more detailed requirements and design definition, ultimately reaching code development. On the right hand side a corresponding set of verification and validation steps confirm the code complies with each layer of design. The lifecycle allows the development of a structured approach to software engineering, well suited to large scale industrial production.

The principal issue with this approach is the delay of verification activities to the so called 'right hand side' of the 'V'. The later in the software development life-cycle a problem, issue or bug is identified in a software system, the more costly it is to fix. This was illustrated by the Constructive Cost Model, COCOMO [8], which showed it typically costs 20-100 times more to fix an error identified after delivery than during requirements definition.

An improvement to the V model is that of the 'W' model, originally introduced by Herzlich [9] as illustrated in Figure 1.2. This model fol-

lows a principal of verifying each step of the software design as soon as the requirements or design have been produced. This allows issues to be identified as close to development as possible, with a view to fixing problems before they propagate through the software design.

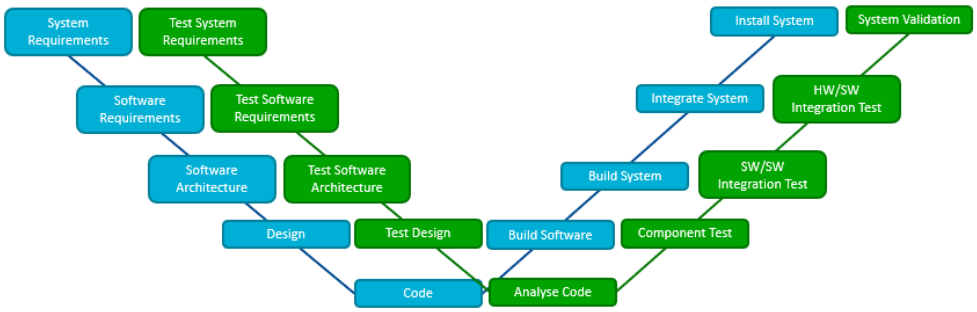


Figure 1.2: Typical Software Development 'W' lifecycle.

The key to this early verification process is not necessarily to produce certification evidence, as this is arguably only produced on the right hand side of the 'W'. The key instead is to identify bugs and issues as early as possible, ideally as soon as a developer has produced a requirement or design. This focus on quickly and iteratively developing quality software is a cornerstone of agile software development techniques[10]. However, in order to facilitate this developers require the right tools, processes and infrastructure to perform software analysis.

Being able to perform software analyses early in a design lifecycle forms a key requirement for the work researched in this thesis. Issues are identified with existing techniques for measurement based timing analysis; which are built on the assumption that the data to drive the analysis methods already exists. Chapter 3 explores the development of automated techniques to aid these processes.

## 1.2 Assessing a Component's Timing Behaviour

One of the key properties of a software system that must be analysed for certification is the worst case timing performance of the system. Typical avionics applications will be designed according to a number of timing related requirements, for example *a response to a specific event shall be completed within a set time.* In order to prove compliance to such requirements the timing performance at the system level must be assessed. This assessment uses schedulability analysis, which requires as a key input an understanding of the Worst Case Execution Time (WCET) of every component within the system.

There are two principal methods for analysing a software component's WCET; static and measurement based.

Static analysis takes the code of the System Under Test (SUT), analyses the possible paths through the code, and by modelling the target hardware; calculates which path through the SUT will produce the WCET. The analysis gains from being able to fully examine the full set of paths through the SUT. However, the primary drawback of static analysis is the technique's reliance on accurate processor models. As developers look to use ever more complex processors; the complexity, portability and potentially pessimism of these models increases accordingly [11].

Measurement Based Timing Analysis (MBTA) approaches rely on measuring the execution of the SUT to provide measured times which are then used to derive WCET bounds. The advantage of this approach is that times can be derived from the actual target hardware, with no reliance on complex timing models. However, the technique suffers from the fact that the software must be executed on the target hardware (or

equivalent cycle accurate simulator) with a sufficient level of coverage to provide accurate results.

In practise, if robust and accurate processor timing models can be developed then the use of purely static analysis methods should provide safe results. However, it is arguably not cost effective to generate accurate timing models of even the simplest processors in use today [12]; including in the industrial system used throughout this thesis [13]. Therefore, the focus of this thesis is purely on measurement based WCET analysis techniques and how they may be improved and automated for use by complex industrial programmes.

Traditionally one measurement technique used in industry has simply been to time the SUT as it is executed as part of standard software verification tests. The maximum observed execution time (MOET, or High Water Mark - HWM) is then taken forward with the addition of a safety bound (defined through engineering judgement) to produce an acceptably sound WCET [14]. Were a system developer able to obtain full path coverage across a system under test, then this approach could potentially provide a reliable WCET. However, obtaining full path coverage quickly becomes infeasible as system complexity increases, and so the biggest risk with this approach is that the testing may not drive the worst case path, producing an optimistic result. Ultimately, even the application of an engineering judgement inflation factor may not produce a safe result [12].

More recently hybrid measurement tools have been applied within industry [15], [16] which aim to address potential optimism in MBTA approaches by combining timing measurements taken during software execution with statically analysed source code information. This reduces the amount of coverage required when producing the timing measure-

ments over exhaustive HWM testing, however the process still requires an extensive amount of coverage [12], [16], [17].

Across the academic literature a number of measurement based WCET tools or processes have been proposed [17], [18]. Again these all simplify the problem, removing the need for obtaining full path coverage. However, they all still assume the method and process for driving the SUT is robust, reliable and sound. In an industrial project this may be generated through software verification activites. Crucially though, this requirement for extensive coverage pushes WCET analysis using a hybrid measurement technique to a late stage in the software design lifecycle. This is less than ideal as it strips system developers of the ability to analyse to provide guidance for optimisation at design time. Furthermore, it delays software timing analysis to a point in the design lifecycle where it is significantly more expensive to resolve issues that arise.

Alternatively, in academia a number of techniques have looked at using the power of automatic test generation to automatically execute an SUT [14], [19], [20]. However, these techniques are assumed to be able to drive the worst case path, which potentially if executed for long enough they will. Unfortunately, when applied to a complex industrial scale system, being developed against tight project timescales, expecting such a tool to obtain a safe WCET in reasonable time is unreasonable. Furthermore, the techniques do not offer support, evidence or validation that the produced results can be relied upon.

Instead this thesis focuses on how automatic test case generation techniques can be tailored to support industrial scale hybrid measurement based timing analysis with a scaleable, portable and efficient approach. Thus allowing system developers to obtain the information they need both to optimise their designs and to facilitate de-risking of software cer-

tification as soon as the software code has been developed. The approach is analysed against a representative set of software components provided by a real avionics system use case, as well as a set of publicly available WCET benchmarks.

## 1.3 Process Proportionate to System

The higher a software component's criticality; the greater the effort involved in the development and verification of the component. In practise, all software components are verified to a certain level; if not for safety critical purposes, then for business or mission critical reasons. This means that even lower criticality components *should* perform as expected. However, the critical point is that these components have not been *proven* to a high criticality confidence level to execute as required. It should therefore be assumed that the information, validation, proof or verification evidence produced for lower criticality components may not provide a full understanding of the component [21].

It is perfectly common for safety critical systems to contain software components with differing safety requirements, and therefore different criticality levels. A good example in most control system architectures is the dual integration of high criticality control and lower criticality monitoring systems. Given that these lower criticality systems have not been *proven* to the same level as the higher criticality components, it is essential for software certification to *prove* that *if* a low criticality component fails, it cannot impact the behaviour, operation or performance of the high criticality component. Arguably it is important to assume the low criticality component *will* fail at some point in time, with an appropriate safety case put in place to protect the wider system.

Traditional software architectures treat each individual processor in a system as belonging to a single DAL, meaning all components executing within that processor must be verified to the highest criticality of the processor. This would lead to an example architecture where one processor may support control software, and one processor may support monitoring components.

This inflexible architecture approach can lead to the introduction of additional processors, even though processor utilisation levels could feasibly allow software co-location. Furthermore, this can also lead to the introduction of significant cross-system communication requirements, potentially adding unnecessary complexity to the system. This approach therefore has the potential to lead to higher development and production costs, as well as increased overall size, weight, and power use.

In the literature a Mixed Criticality System (MCS) is a system which combines software of multiple DALs on the same processor. The technical objective of which is to provide sufficient evidence that a low DAL component cannot jeopardise any high DAL component's temporal or functional requirements, while still providing a level of service to the low DAL component. One approach to MCS development is to deploy the partitioned architecture defined by the ARINC 653 standard [22]. This standard defines a partitioned model principally aimed at the development of Integrated Modular Avionics (IMA), but is capable of supporting partitions developed against different DALs. The issue with the ARINC 653 approach is that the solution defined for temporal partitioning, essentially a two-level scheduler with time division, makes the approach difficult to apply to a complex control system [23]. This is because it can lead to the introduction of higher release jitter, longer end-to-end transaction response times and in general it can be difficult to accommodate

a complex task schedule, including aperiodic operations, into fixed time partitions [23].

Since Vestal's seminal work [24] on the topic of an MCS scheduling a significant number of academic works have been published on the development of MCSs. In particular, much of this work has focused on the temporal partitioning aspects of mixed criticality scheduling [25]–[30]. However, while addressing temporal partitioning, these methodologies do not progress far enough to allow integration of an MCS into a high criticality system. Crucially, the literature does not address how such a system should be designed, analysed, validated and certified. This includes the appropriate handling of overheads, and how to assess the service afforded to low criticality tasks. This is particularly important as in order to deliver a credible product, system integrators may need to provide guarantees on the minimum level of service provided to low criticality functionality.

This thesis aims to address these issues by contributing a design for a MCS, justified against the certification guidelines provided in DO-178C [6]. This includes how such a system should be designed and analysed. Secondly, the thesis describes the application of this MCS to a large complex industrial application, and introduces a process that could be used to study and understand the service afforded to a set of low criticality tasks. The developed system is assessed against the avionics system requirements, certification evidence and performance at the system level. In addition, several publicly available system examples are used to assess and review the approach.

## 1.4 Coping With WCET Pessimism

In order to provide sound WCET results to the appropriate level of confidence it is logical, and almost expected, that timing analysis processes will induce pessimism [31]. This can be induced at the task level due to system designs that incorporate infeasible paths, or through analysis deficiencies when viewing code constructs such as loops. This pessimism is extrapolated as schedulability analysis expects each task to execute to its WCET, all on the same iteration. For instance, in a complex control system a reaction to two opposite events (such as reacting to an overspeed and an underspeed) may appear on the worst case path, and be assumed to happen on the same iteration.

Identifying such forms of pessimism when viewed at the macro level is easy. However, when viewed at the system level featuring thousands of functions such identification quickly becomes infeasible in a cost effective time-frame. The system, and key safety critical functionality, must be confirmed to comply with its requirements, even if executed against the worst case. However, if the system is not executing to the worst case, then is it possible to use the spare utilisation generated by WCET pessimism for useful execution?

One possible solution for this would be to execute less time critical operations within an idle task, or as the lowest priority tasks within the system, on the assumption that key safety critical operations will interrupt their execution when required. This offers an easy method for utilising spare execution time, however, the process provides no ability to add timing constraints against the software executed in the idle time. Where it is desirable for less critical components to be developed against temporal requirements, a more advantageous approach would be to utilise some of the techniques introduced by MCS schedulers.

Such approaches, for instance [25]–[27] allow less critical components to execute within a system, provided that high criticality components execute within certain, less pessimistic, possibly optimistic, timing budgets. The techniques introduced in these papers provide the facilities to prove adherence of high criticality tasks against their timing requirements, this is based on the assumption that when necessary all low criticality tasks will be disabled. Extensions to these models, such as [32]–[34], employ methods such as elastic scheduling or graceful degradation to attempt to extend service provided to low criticality components. However, at present all of these techniques lack mechanisms to assess the service provided to low criticality tasks within such a system. These low criticality tasks will still have some system function, even if not a safety critical function, and therefore understanding how they execute in practise is still important from a system verification point of view.

This thesis aims to address this by identifying an iterative process, based around a system simulator, that aims to provide a mechanism for assessing the service afforded to a low critical task. The process is broken down using a Goal Structuring Notation (GSN) [35], before being applied to a real industrial case study.

## 1.5 Difficulties of Applying Academic Research to Industry

Sections 1.1, 1.2, 1.3 and 1.4 have introduced some of the real-time software development problems facing industry today. There are processes and methods in the published academic literature that have the potential to aid industry; these include the application of automatic test generation and preemptive mixed criticality scheduling. However, these techniques

have yet to be tested and analysed in large scale industrial projects, and as noted by Quinton [36] and Davis et al. [15] there are significant difficulties that face the application of academic research in industry.

These difficulties in part come from the fact that there are few robust industrial scale examples that can be used to develop and test techniques and processes at the scale required for industry [36]. This has in some cases led to academic research that focuses on solving problems not found in industry, or alternatively research that focuses on new problems, assuming the problems of yesterday have been solved.

This thesis aims to advance and extend the academic research already available in the literature and to examine how it can be applied to real industrial applications. The key contribution that encompasses all of the work in this thesis is in the application of the reviewed and updated approaches to a real industrial mixed criticality application, with no assumptions or simplifications made to the system being studied. The system used for this analysis is introduced in Chapter 2; consisting of a control system taken directly from a Rolls-Royce aircraft engine.

## 1.6 Thesis Proposition

The central proposition of this thesis is:

*Automatic test case generation techniques can be extended to reliably target hybrid measurement based timing analysis to produce sound WCET profiles. These produced WCET profiles can then be used to aid the development and validation of mixed criticality schedulers, provided the certification objectives, overheads of the scheduler, and the service provided to low criticality tasks are not neglected.*

The key strands to this thesis are as follows:

- To study the application of automatic software execution towards industrial scale hybrid measurement based WCET analysis. *The key contributions being an algorithm designed to provide the timing measurements required by a hybrid measurement based WCET analysis tool, as well as results from applying the technique to an industrial scale case study.*

- To identify how a mixed criticality system may be developed and certified. *The key contribution being a design for a mixed criticality scheduler, overhead analysis and run-time monitoring system defined according to certification standards.*

- To show how an existing industrial scale project may be ported to a mixed criticality system. *The key contribution being a process for automatically porting an existing system to a preemptive system while minimising system overheads.*

- To present a coherent system development process that uses the defined automated WCET analysis techniques to develop and validate the mixed criticality system. *The key contribution being a process for validating the service afforded to a low criticality task within a mixed criticality system.*

## 1.7 Thesis Structure

**Chapter 2** introduces the current industrial system that is used throughout this thesis. The tooling, techniques and processes researched in this thesis have all been adapted and applied to the DAL-A aircraft engine control system introduced. Several different variants of the control system are used throughout this thesis, in all cases without modification,

and in some cases as part of a live development project. This forms one of the central themes of this thesis - the processes, techniques and tooling studied are tested and applied to a real certified system in order to test their effectiveness and applicability.

**Chapter 3** considers how the timing performance of the system may be analysed automatically using measurement based approaches. The chapter assesses possible options for automatic software execution from the available literature on automatic test case generation. It discusses the development of a set of algorithms designed to automatically drive a software component to produce the required timing information; before applying each algorithm, as part of a comprehensive statistical evaluation, to a set of components provided by the system defined in Chapter 2.

**Chapter 4** studies the definition of an industrially appropriate, certifiable, Mixed Criticality System. The chapter begins with a review of the existing literature. It then advances to identify the key requirements from a certification point of the view for the system, before researching and assessing how such a system can be designed and verified. It then progresses to assess the most efficient way of porting the system described in Chapter 2 to the new scheduler. In particular, this involves the porting of the non-preemptive system to the fully preemptive mixed criticality scheduler. The system's static schedulability is analysed, along with an exploration of the benefits of the new system.

**Chapter 5** extends the system developed in Chapter 4, and defines a process that would allow the produced system to be validated. This includes the development of a new process to examine how the service provided to low criticality tasks within the system can be assessed; a process which utilises the timing analysis tooling researched in Chapter

3. Finally, this process is applied to a set of case studies for the system in question in order to assess the process' applicability and effectiveness.

**Chapter 6** summarises the work conducted in this thesis, and provides guidance for future research in this area.

# Chapter 2

# The Industrial Context: A Current FADEC System

This thesis focuses on the industrial application and extension of advanced real time systems research. As such it discusses the development of a single industrial target: a high criticality Rolls-Royce aircraft engine control system, or Full Authority Digital Engine Controller (FADEC). The system used throughout this research is analysed directly from project, with no simplification or modification.

As introduced by [15]:

*FADECs are responsible for the control and monitoring of aircraft engines. They play a vital role in not only the reduction of hazardous events related to the aircraft engine, but also the overall safety and certification of the aircraft. FADECs do much more than inject fuel and control the engine. They help keep both the aircraft's cabin and fuel at the right temperature, receive information and commands from the cockpit and send back information, they also log information about the engine for future maintenance, and play other vital roles such as helping the aircraft brake on landing via the use of thrust reversers. Over time, this has led to an*

*increase in the amount of software in the system, most of which is hard real-time.*

The Rolls-Royce FADEC architecture is currently going through the most ambitious redesign in over 30 years. The new FADEC architecture is being updated in order to support the Rolls-Royce UltraFan$^{TM}$ engine architecture, in itself the greatest aircraft engine core architecture change in 60 years. The UltraFan$^{TM}$ engine will introduce a powered gear box into the centre of the jet engine, and will require a significant step increase in software system size, with the new control system estimated to be several times larger than existing FADEC systems. This is within a climate where software development cost is already considered a significant problem [1].

The following sections now explore the current system's WCET process, target processor, scheduling methodology and architecture.

## 2.1 Current Approach to WCET

The FADEC aircraft engine control software, which is written predominantly in the SPARK 95 subset of Ada; consists of several hundred individual tasks formed by several hundred thousand lines of code. Each software component is analysed using the hybrid measurement based tool RapiTime, from Rapita Systems Ltd.

RapiTime automatically instruments the system source code and analyses the structure of the code. When the instrumented code runs on the target; the instrumentation produces a timing trace that is then analysed off-line and together with a high-level structural analysis to produce a timing profile for the software.

The Rolls-Royce approach to software certification using RapiTime,

as described in [13], is to integrate the tooling with the low-level software verification process. This allows timing measurements to be taken as software verification is performed, delivering sound results in time for certification.

This process, however, while being used successfully to certify several projects since 2015, is far from ideal from a cost effective point of view. The principal issue being that the generation of accurate timing data is left to a point in the design lifecycle which is too late for cost effective optimisation. This has had the consequence of separating system developers from the optimality of the code they produce.

One approach to improve this may be to require each engineer to manually execute the code they have just produced. However, this approach is less than ideal for two reasons. Firstly, as the WCET of each function must take account of each function it calls, in a complex control system, this approach quickly becomes infeasible as the size of the test space increases. Secondly, each tester would have to derive a test that provides the appropriate coverage of the whole system-under-test that is required by the hybrid measurement based analysis tool. On the one hand this is a process that could also allow the early identification of software bugs. However on the other; a software developer would be expected to develop a greater number of test drivers to generate enough system coverage, than they would have to produce purely to debug their code.

## 2.2 Target Processor

The target processor used throughout this thesis is the Rolls-Royce in-house processor. The Rolls-Royce processor is a packaged device that

integrates a core, memory, IO and tracepoint interfaces. Being targeted at the safety-critical embedded sector, the device is DO-254 – Level A compliant. It has extensive single-event-upset protection and is suitable for harsh environments. The processor does not incorporate a data or instruction cache due to their impact on timing predictability.

The processor has been carefully designed to ensure that each instruction's execution is time-invariant. In other words each instruction will take the same time to execute, regardless of the data its operation is performed upon. These design features further ensure that previous processor state has no effect on the current operation of the device. The use of such a deterministic processor allows worst case timing measurements of software components, including the scheduler, to be taken during normal operation, without the need for special builds [13], [37]. Finally, the processor provides the facility for implementing user and supervisor mode memory partitioning.

This processor is targeted throughout this thesis as it provides a real example of a processor in use in both current and future avionics applications. However, an implicit requirement of this research is to keep the developed techniques platform independent, although such independence is not explored further within this thesis and is saved for future work.

## 2.3 Current Scheduling Approach and Architecture

The current scheduler used within the FADEC software system is a fixed priority non-preemptive scheduler; the initial development of which is discussed in [38]. The system, and each component within it, must be carefully developed in order to avoid long blocking terms and excessive

scheduler overheads. The scheduler was developed to the highest criticality standards, against DO-178C, and has been in use on all Rolls-Royce FADEC systems for almost 20 years.

An important aspect of the system, and its associated schedulability analysis tooling, is the use of a repeatable algorithm (i.e. one that always produces the same results) that takes all the temporal requirements of each task and uses them to calculate a deadline for each task. Task priorities are then assigned using the Deadline Monotonic Priority Ordering (DMPO) algorithm where the task with the shortest deadline is given the highest priority. If all deadlines are met, then all the timing requirements are met; the method ensures the schedule is correct by construction. This approach has a further advantage, key to industry, that by incorporating the timing requirements for each task into its design-time calculated deadline; the system can easily be proved, reviewed and understood by engineers and system integrators [39].

The current FADECs designs consist of a large number ($> 200$) of tasks. Because of the real-time, hardware controlling nature of the control system, a number of tasks (in the order of 5% of the total number of tasks) have completion jitter requirements. To comply with their requirements, these tasks must execute within the *jitter_requirement* of their period. Typically these tasks tend to take on the highest priority (lowest deadline) across the system.

Furthermore, in order to prove adherence to system level temporal requirements, such as the system's response time to certain engine events, the control system task set has been designed to incorporate a large number of transactions. A transaction is a sequence of tasks that must execute in a defined order. Transactions can contain sets of tasks with different periods, and tasks that are defined against a jitter requirement.

Together a task's jitter requirement, transaction requirements, and period form the set of temporal requirements which are used to define the deadline of the task.

At present all tasks within the system are defined as high criticality DAL-A tasks, and so all tasks are designed and proven against the most stringent development standards. Furthermore, all tasks are treated as hard real time tasks. In practise, some tasks, assuming a carefully orchestrated safety argument could be made, could be treated as lower criticality tasks, or indeed as soft real time tasks. This however is not currently possible without an appropriately designed mixed criticality scheduler, appropriate system level partitioning and a robust validation of the service provided to any tasks treated as soft real time and/or lower criticality tasks. The available literature does not yet provide such guarantees and validation processes.

This is important because it is not simply acceptable to assume that soft real time tasks and/or low criticality tasks can be disabled for extended periods of time. The tasks still have a business critical operation; even if they do not have a safety critical one. In essence, a task's criticality is not necessarily related to its 'importance'. Therefore, it is only possible to consider mixed criticality operation, if as part of a system certification, or mission validation effort, the service provided to low criticality tasks is understood.

## 2.4 System Model

A task, or partition, is a schedulable entity which consists of a number of components. This task completes a system functionality which carries a failure condition. This failure condition reflects the system-level effect

that a failure of this task may cause and leads to the derivation of a DAL, also referred to as a criticality level. The criticality level of the task is denoted by $L_i$ where (as defined by DO-178C [40]) $L_i \in \{A, B, C, D, E\}$. Level A indicates the highest level, E indicates the lowest level such that $A \geq B \geq C \geq D \geq E$.

A system is defined as a collection of tasks denoted by $\tau_i$ where $1 \leq i \leq N$. Each task $\tau_i$ is denoted by a **deadline** $D_i$, a **period** $T_i$, a **criticality level** $L_i$, and one or many **WCETs** $C_i$. A task is said to have a hard deadline if the task must complete before said deadline; whereas a soft deadline allows deadlines to be missed without having an adverse impact on the safe operation of the component. The current system model assumes if a task exceeds its deadline it is permitted to continue to completion.

Other parameters which describe a task include the completion **jitter** $J_i$ which denotes the maximum permissible variation of the period $T_i$ for the completion of the task. Once a task has been scheduled it may be assigned a **priority** $P_i$ where $1 \leq P_i \leq N$. It is possible for the execution of one task $\tau_i$ to be reliant on the completion of another task $\tau_j$. Such an interaction is described as a **transaction**. Transactions are formed in order to aid the proof of system-level timing requirements, where it may need to be proven that the system performs a set sequence of activities in order, and within a set interval of time. The maximum response time of a task $R_i$ is calculated as the sum of the WCET $C_i$, the **interference** suffered by the task $I_i$ and the **blocking** suffered by the task $B_i$; where the Interference $I_i$ is the sum of the time delay between release of the task, and execution as caused by higher priority tasks. The blocking time $B_i$ is the time delay between release of the task and when the task begins to execute, as caused by lower priority tasks.

Finally, a hard deadline task $\tau_i$ is said to be schedulable if its **worst case response time (WCRT)** $R_i$, is less than or equal to its deadline $D_i$.

## 2.5 Summary

This chapter has introduced the industrial context, from the point of view of a real aircraft engine control system, or FADEC. The system is introduced in order to provide a sound base to assess the research conducted in the following chapters.

Introduction of the system has already presented a number of research challenges that are revisited in the following chapters, these include:

- How to efficiently assess the timing properties of a complex system encompassing thousands of different functions developed over a multi-year programme.
- How to appropriately schedule complex task sets, including the appropriate handling of transactional and jitter requirements.
- How to provide assurances that all tasks (including low criticality tasks) comply with their requirements.

# Chapter 3

# Obtaining Reliable Task Timing Profiles

The understanding of, and confidence in, a software component's WCET is a key validation step that must be completed during the verification and certification of a safety critical system. DO-178C dictates that a system developer should understand the worst case timing behaviour of the system, and be able to provide confidence that any timing requirements in the software design have been complied with in the implemented system. One principal method for ensuring compliance to timing requirements is to analyse the WCRT of each task, which in turn requires a bound on the WCET for each task to be identified. This WCET analysis should take account of any performance effects introduced by either the compiler, or any advanced hardware features. The analysis should also provide an understanding of the timing behaviour of the task within the integrated system with all inter-system timing impacts taken into consideration. Ultimately, this worst case behaviour must accurately reflect the performance of the system in service.

In practise, even on a simple processor analysis of a program's WCET

can become an extremely difficult and expensive process to perform to an industrial scale, frequently requiring significant engineering effort. This chapter explores how data to support hybrid measurement-based WCET analysis can be reliably generated automatically to help mitigate this problem.

Section 3.1 now explores the available literature on both measurement based timing analysis, and on automatic test case generation. Section 3.2 examines the target application for timing analysis, before Sections 3.3 and 3.4 explore an automated timing analysis approach built upon optimisation algorithms. The approach followed in Section 3.4 begins by examining the effectiveness of a purely random optimisation algorithm (or search), as well as the application of the current approach found in the literature. The section then progresses to examine how this algorithm can be improved and refined using a series of examples taken both from industry and from openly available benchmarks.

## 3.1 Literature Survey

Throughout the literature survey discussed in this section the actual-WCET is assumed to be unknown. The term *accuracy* is used to denote a WCET approaching the actual-WCET of the system in question, whereas a *sound* WCET is used to denote a WCET which rests above the actual-WCET. The aim of a timing analysis process should be to provide a sound WCET, with acceptable accuracy to firstly avoid undue pessimism and secondly to provide a real representation of the final target system timing performance. Finally, any industrial scale process must be efficient and repeatable to allow affordable large scale application.

### 3.1.1 Measurement-Based WCET Techniques

Industrial techniques in the past have centred on the simple process of taking High Water Mark (HWM) timings from software test executions [14]. This approach is easy to implement and guarantees to provide measurements of the real system; and provided the System Under Test (SUT) executes full path coverage it should provide a sound result. However, should the testing not provide full path coverage then this approach risks producing optimistic results as the software may not execute the worst case path. As a system expands, the number of possible paths increases accordingly, and the number of tests required to obtain this coverage can be assumed to grow at a similar rate. In essence, in a complex software program the possible search space, which includes the worst case path, increases to such a size that this kind of analysis becomes unsound and infeasible.

Research targeting sound and affordable measurement-based timing analysis has taken two main paths; hybrid approaches, and probabilistic approaches. Hybrid approaches combine structural information obtained through static analysis, to measurements taken during execution. In contrast, probabilistic approaches apply statistical theorems over a large number of execution time measurements to produce a probabilistic distribution of execution times; ultimately producing a WCET value against an expected exceedance probability.

Hybrid measurement techniques aim to simplify the execution time measurement search space to an affordable, achievable and practical level; while improving the safety of general measurement-based approaches. Deverge & Puaut [17] for instance use structural analysis to condense the SUT into a number of clusters. Each cluster is then analysed individually, the aim being to achieve full path coverage through the cluster.

Each path is then timed to produce an observed WCET for the cluster. These cluster-WCETs are then combined using data extracted from the structural analysis phase to produce a final WCET.

The method relies on the assumption that each cluster is context independent from all other clusters. The authors suggest three areas where this assumption is broken - Global Mechanisms (cache, branch predictors, etc), Variable Latency Instructions (instructions with variable timing behaviour, e.g. integer multiplication, or FPU operations) and Statistical Execution Interference phenomenon (due to delayed memory accesses or memory operations) [17]. It is suggested that in order for clusters to be handled independently, these three causes of processor unpredictability must be mitigated.

Stattelmann & Martin [18] present a measurement-based tool that also breaks the SUT into a number of easily traceable segments. However, they overcome the requirement that each segment be independent by formulating the WCET as a product of its execution history, or context. The SUT is divided into a number of program segments, which are executed on hardware and analysed by trace hardware. The context-sensitive evaluation relies on the tracing hardware being able to consider the execution history prior to a run of the segment. This execution history is extracted as the first part of the code segment is executed and added to the Control Flow Graph (CFG) of the program segment under test [18].

Once all the trace data and execution history has been extracted, the execution times for each basic block are annotated onto each node of the CFG, but only where the node's execution context matches. This produces a context dependent set of times for each basic block. The execution history is then traced through the CFG to identify the path

containing the largest execution time [18].

The tool developed as part of [18], was tested on the Mälardalen WCET Benchmark Suite [41] and the DEBIE-1 benchmark. It showed how the context dependent measurements were able to obtain results higher that the maximum end to end observed times of a longer run, but also lower than results obtained through non-context dependent analysis.

Petters [42] proposes a process that uses source code instrumentation to target analysis, measurement and specific paths through the code in order to produce context sensitive manageable blocks for analysis. These block times are then rolled up to produce a system level result. One risk with this approach is that the instrumentation and measurement control is inserted into the code, which can be expected to affect code execution and compiler optimisations, meaning the analysed system may not reflect the final un-instrumented system.

Ultimately though, the risks with each of these solutions is their scalability; because as the complexity of the system increases the number of sections the code is broken down into would also increase accordingly. For a large industrial scale project this could lead to tens of thousands of functions all being analysed to provide path coverage, or all producing a context-sensitive WCET equation, and so the processing, or engineering effort required would be significant. For instance; the current Rolls-Royce control system introduced in Section 2 consists of over 5000 functions, executing over 250,000 lines of code.

The RapiTime tool from Rapita Systems is a commercially available hybrid measurement-based timing analysis tool, which is already in use in industry [13], [15], [16]. RapiTime statically analyses the source code of the SUT to obtain a tree based representation of the code. The tool then observes the execution of the software as it executes on target hardware

and appends this timing information to the tree based representation [43]. This provides a time profile for each basic block through the code, which is then combined in a final calculation stage to provide a WCET estimate. The key is that this breakdown to basic block level is hidden from the user of the tool.

As the analysis is built around measuring the actual execution time of the target hardware, the tool is easily ported to new hardware and software architectures. However, this also means the tool requires a pre-defined comprehensive test set to properly drive the SUT, ensuring enough coverage is obtained to generate accurate results [44]. This means that the tool cannot be applied to a system until a point where verification activities have matured, therefore the analysis is delayed until a later, more costly, point in the design lifecycle.

Finally, Measurement-Based Probabilistic Timing Analysis (MBPTA) was first proposed by Stewart & Burns [45]. This was later extended by Hansen et al. [46] and Cucu-Grosjean et al. [47]. The basis of these techniques is the use of Extreme Value Theory to fit an appropriate distribution to the observations captured. The WCET is then extracted from the distribution for a chosen level of probability that it has exceeded. The problem is that in order to provide reliable results the input data fed into the tool must be independent and identically distributed, which in practice is hard to achieve. Secondly, the level of code measurements required, in some cases branch, decision and state coverage, makes the problem of obtaining reliable measurements to support this type of analysis even harder to solve [48].

### 3.1.2 Garbage in, Garbage out

Accurate and sound measurement-based timing analysis tooling, regardless of the approach utilised to simplify the problem, is wholly reliant on the data input into the analysis [11]. If insufficient or inadequate timing information is input into the process, then the results that are produced by the tooling may not provide a sound, let alone accurate, result. Additionally, as more complex processor architecture features such as caches are considered, the amount of timing information and data required to provide a sound WCET can be expected to increase even further [12]. Particularly, due to the introduction of timing anomalies [49].

Colin and Petters present an investigation into the effects that different advanced processor features have on the WCET and on its probabilistic analysis [50]. The features investigated include data and instruction cache, branch prediction units and out of order execution units. The paper's aim is to show that benefits in performance offered by the advanced features can be seen in the WCET analysed on these architectures.

The analyses performed, over five complex algorithms, tested ten configurations with and without the advanced processor features under test. Tests were performed using randomly generated test vectors. The investigation in [50] utilised the pWCET tool [51]; the results of which show an interesting correlation between advanced processor features and timing improvements. However, it also highlights how the advanced processor features tested significantly increase the complexity of achieving enough test coverage to obtain accurate results.

Betts et al. attempt to address issues concerning coverage by introducing a new concept for measuring WCET coverage for measurement-based approaches [44]. Their coverage metric is based on an amalgamation of three approaches which, when combined, form the basis for

approving that enough test data has been produced to support dynamic analysis.

The principle of building an instrumentation point graph (IPG) is introduced. The graph splits the SUT into a number of execution units, divided between instrumentation points. The IPG details the execution unit's interactions between these points, and this forms the basis for the metric's algorithm. The three metrics introduced in [44] are firstly Simple Pipeline Coverage, which measures that every execution unit between instrumentation points has been executed. Secondly, Pairwise Pipeline Coverage, which measures that every node into each instrumentation point, and every node out of each point has been tested. And finally Pipeline Hazard Path Coverage (PHPC) which measures that every statically defined pipeline hazard has been observed at least once during testing.

This final metric PHPC is perhaps the most difficult to define and indeed fulfil. The metric is reliant on static analysis of the SUT to identify potential structural and data dependent pipeline hazards, which of course requires detailed knowledge of the pipeline the SUT targets, therefore affecting the portability of the tool. The process of identifying the SUT's pipeline hazards through static analysis is also not a trivial task. To fully achieve this would not only require knowledge of the current instruction's effect on the pipeline, but also on the current and previous state of the pipeline. Arguably this level of analysis would quickly prove infeasible if applied to a large industrial system.

Ultimately, the previous work discussed in this section has explored how confidence can be built that sound WCETs have been obtained through measurement based coverage. However, each method assumes that the measurements required to drive analysis already exist, without

addressing how the measurements can be produced. With respect to the requirements placed on these measurements; techniques that try to break down the SUT into sets of sections to be analysed exhaustively, such as [17], [18], risk scalability or in the case of [42], correctness. In some cases it is unclear what coverage is required to obtain a sound result, such as in the case of probabilistic timing analysis techniques [46], [47]. Finally, extended coverage metrics have been proposed by [44], [52], however these risk requiring infeasible levels of coverage.

### 3.1.3 Producing Measurement Data to Support WCET Analysis

Optimisation algorithms, specifically search algorithms, are designed to iteratively and efficiently improve on a defined solution through extensive trial and error. Wegener [19] and Tracey [14] both illustrate how search algorithms could be used for test data generation.

Wegener's early work [19] built off Jones et al. [20] and presented an investigation into how genetic algorithms can be used to estimate the minimum and maximum execution times of software targeting embedded systems. Tracey introduced a framework of tools designed to automatically generate test data to perform dynamic analysis on an SUT. One of the targeted analyses being the analysis of the WCET. The work has been targeted toward safety-critical systems using strongly typed Ada [14]. The framework introduced is primarily based on search algorithms, which produced good results when compared to system HWM observations. However, the drawback was that the tool had to achieve path coverage to obtain a sound and accurate WCET.

Khan and Bate [53] introduce the idea of incorporating multi-criteria optimisations into a search based WCET analysis tool. The method

adopted used a number of fitness function parameters in order to attempt to drive the worst case path. These included advanced processor features known to cause larger WCET values, such as cache misses, but also focused in on low level software coverage such as loop iterations. The paper concluded that no one fitness function provided better results across all test code items, and that the fitness function chosen should be dependent on the target environment. However, the paper focused on a number of processor or software features that are not necessarily present in safety-critical systems and also failed to consider coverage which is of importance to certification. Nevertheless, the work did indicate that using optimisation algorithms focused on the features that contribute to higher-WCET figures could produce more reliable results.

Williams [54] proposes a static analysis tool which aims to identify a test vector to exercise every path through the code under test. The WCET can then be read off as the HWM observed during testing. This was extended by Williams and Muriel [55] with an analysis into possible simplifications that can be made to avoid the analysis requiring full path coverage. This includes maximising loop counts and assuming branches are always taken. The paper recognises that further investigation and justification is required, however it does indicate possible areas where MBTA coverage requirements could be simplified.

Wenzel [56] introduces an MBTA tool designed to calculate sound WCET bounds of safety-critical software. The tool uses a combination of static analysis and dynamic measurement of the SUT in order to compute sound WCET bounds. It statically analyses the feasible paths through the code and then uses search algorithms to identify test vectors to execute each path. This is achieved through a combination of test data reuse, random search, genetic algorithms and finally model checking [56]. Un-

fortunately the tool places a number of restrictions and assumptions on the code under test; for example the tool is only capable of analysing acyclic code and does not allow function calls. This means that unfortunately the compromises required to use the tool are significant, and would not be acceptable in an industrial environment.

Building off this Bünte et al. [52] examined the effectiveness of using model checking [57] to produce test suites with enough coverage to provide reliable WCET estimates once combined using Implicit Path Enumeration Technique (IPET). Their research focuses on identifying effective coverage metrics to drive a model checking test suite generator - the so called FORTAS framework. This was extended by Bünte et al. [12] where the research combines the results produced with a genetic algorithm, which then aims to identify larger execution times. One drawback is that the tool analyses software that has been simplified to ensure each decision point relies on only a single variable. Furthermore, some of the benchmark tests utilised had to be simplified to allow analysis using the bounded model checker [52]. This may not be appropriate to an industrial program where the cost of simplifying hundred of thousands of lines of code could make this process infeasible.

This work was further extended by Kirner et al. in [58], [59]. Their initial work [58] examines how compiler optimisations affect source to object code traceability, and therefore which optimisations affect the test vectors produced by the model checking test suite generator tool. It defines which optimisations need to be turned off in order to guarantee source to object traceability for various code constructs. The paper also examines the effect that turning these optimisations off has on a test processor (Intel Core 2 Duo) for a limited set of Mälardalen benchmarks [41]. Finally, their later work [59] expands on the FORTAS toolset with

the implementation of context-sensitive-IPET.

The FORTAS framework represents some of the most advanced work in the field of generating traces for measurement-based timing analysis tooling in the available research. However, its scalability remains limited by its bounded model checking tooling. Furthermore, the restrictions placed on the source code being analysed, including code simplification and compiler optimisations, may limit its usability. However, if these can be addressed then the tooling and techniques offer potentially analogous tooling to those discussed in this thesis.

### 3.1.4 Summary of the Literature Surrounding WCET Analysis

A review of the available literature has illustrated some of the key issues surrounding WCET analysis. As discussed in the introduction to this section, an industrial WCET tool should be expected to provide sound, accurate results efficiently. These results should be able to be produced and reproduced as part of an affordable process that ultimately provides a system designer with confidence in the result.

Techniques that have focused on soundness and accuracy, risk unacceptable scalability and inefficiency. Conversely, techniques that focus on efficiency, risk poor accuracy and incorrectness. Whereas techniques that restrict the target application to improve WCET tooling efficiency, risk leading to unacceptably expensive product restrictions. Ultimately, a compromise of these different requirements must be sought.

This thesis chapter is concerned with using search algorithms to generate good data for input into Measurement Based Timing Analysis (MBTA) tools. The general proposition is that a search algorithm, or indeed any test data generation technique, cannot be expected to stum-

ble across the WCET of a software component if that is its only target. Instead the optimisation algorithm should be focused on generating the *right* coverage to support sound measurement-based WCET analysis.

This allows the search algorithm to be focused on a smaller, more manageable search space that delivers the 'good input data' required by the timing analysis method adopted. The work differs from previous approaches, such as the work of Wenzel [56] and Bünte [12] as firstly the fitness functions used have been specifically tailored to target the type of data needed by the MBTA tool. Secondly, while the approach has been designed to analyse industrial software developed and reviewed against strict standards, the analysis places no further restrictions on the software under test (unlike approaches such as the FORTAS framework [12], [52], [58], [59]). Finally, the approach has been investigated on a processor, and software set taken directly from an industrial system. This includes software that incorporates a large amount of previous software state, which significantly increases the search space.

## 3.2 Target Application

In order to study the application of automatic test generation techniques to the derivation of measurement-based timing data, a set of test code items was compiled. Aiming to provide a broad subset of examples covering the principal architectural components found in real systems, the test code items were derived from both the Rolls-Royce system defined in Chapter 2 and from the Mälardalen WCET benchmarks [41].

The Rolls-Royce test code items introduced in Table 3.1[1] consist of a set of complex 'high level' software components with a considerable

[1]The acronyms used for the Rolls-Royce test code items are not expanded as their full name may reveal commercially sensitive information.

Table 3.1: Test Code Items Used for the Analysis.

| Name | Source | Loops | LOC | MCC | Inputs $I/F/B/S^I/S^F/S^B$ |
|---|---|---|---|---|---|
| QSort | Mälardalen | Y | 121 | 21 | 0/20/0/0/0/0 |
| Qurt | Mälardalen | Y | 166 | 19 | 0/3/0/0/0/0 |
| Select | Mälardalen | Y | 114 | 20 | 1/100/0/0/0/0 |
| InsertSort | Mälardalen | Y | 7 | 5 | 100/0/0/0/0/0 |
| F | Rolls-Royce | Y | 1101 | 154 | 0/17/12/194/32/24 |
| ACDF | Rolls-Royce | N | 85 | 9 | 0/7/4/16/0/6 |
| ACDN | Rolls-Royce | N | 167 | 14 | 0/6/6/15/2/8 |
| ACDP | Rolls-Royce | Y | 254 | 27 | 0/8/5/16/0/6 |
| ACDT | Rolls-Royce | Y | 395 | 55 | 0/26/13/48/0/18 |
| VCA | Rolls-Royce | Y | 590 | 68 | 1/40/17/9/6/6 |
| VCP | Rolls-Royce | Y | 922 | 94 | 1/44/43/10/11/9 |
| VCS | Rolls-Royce | N | 205 | 21 | 0/6/2/0/0/0 |

number of inputs, which collectively control the operation of the code. Table 3.1 denotes whether a test code item includes Loops and the total number of executable lines of code for each item (LOC). The McCabe Cyclometic Complexity (MCC) metric [60] is shown to provide a range of how many independent paths there are through the code. Finally, the number of inputs that drive the test code item are shown as $I/F/B/S^I/S^F/S^B$; denoting each input variables type as $I$ = Integers, $F$ = Floats, $B$ = Booleans, $S^I$ = State Integer, $S^F$ = State Float and $S^B$ = State Boolean. Where State variables are parameters that are both read from, and written to by the test code item, for instance, these variables could hold a previous system parameter for comparison on the next iteration of the control code.

The four standard benchmarks used for the analysis were taken from the Mälardalen WCET Benchmarks [41]. A large number of the benchmarks were not included as they provided constant execution times when executed on the target processor and hence were not sufficiently interesting. The benchmarks used were chosen as the execution time of each varies significantly as the input search space is traversed, and because they contain input data dependent loops. Finally, two of the chosen Mälardalen benchmarks (InsertSort and Select) were extended to create a larger search space. In both cases the number of input variables was increased from 10 to 100.

## 3.3 Optimisation Algorithms

The search algorithm used for the analysis is a derivative of the simulated annealing algorithm, originally presented in [61]. The basic algorithm is shown in Algorithm 1.

The simulated annealing algorithm was chosen over other algorithms, such as a genetic algorithm, because of its ability to narrow down on a good solution, while also searching over a large part of the search space. Although the key to this work is the fitness functions proposed; there is no reason why these fitness functions could not be used to drive a genetic algorithm.

On each iteration the *GenNewSolution* function pseudo-randomly selects a new input solution to the function under test. This solution is generated from the previous solution, with only a minor change to a single randomly selected variable. *FitFunc* is then used to assess the new solution's fitness, which is accepted by the *if* statement on line 5, if an improvement, or pseudo-randomly selected or rejected if a degradation.

As the test progresses the pseudo-random selection of worse solutions will reduce, as controlled by $Temp$. Finally, $StoppingCriteria$ assesses whether to stop the search.

**ALGORITHM 1:** Simulated Annealing.

```
1:  Temp = [0.01, 0.1]
2:  while NOT StoppingCriteria() do
3:     NewSolution = GenNewSolution(CurrSolution)
4:     Fitness = FitFunc(TestCode(NewSolution))
5:     if random(0..1) <exp(Fitness / Temp) then
6:        CurrSolution = NewSolution
7:     else
8:        ignore new solution
9:     end if
10:    Temp = CalculateNewTemp(Temp)
11: end while
```

The initial algorithm parameters were defined using an extensive trial and error approach against the following criteria:

- Sufficient exploration of the search space, initially allowing regular solution degradation while ensuring solution improvements are always accepted and pursued.
- Execute for a sufficient length of time, with the initial evaluation target being to execute for significantly longer than necessary.
- Each fitness function should use the same algorithm parameters to ensure fairness and control across experiments.

The search algorithm parameters discussed in the following sections were derived through this approach, and are not discussed extensively as they do not form a key contribution of this work. The key for the

fitness function improvement, discussed later in the following section, is that the parameters aim to provide a fair playing field for studying the different fitness functions that do form one of the principal contributions of this work.

The following sub-sections now discuss the key parameters of the simulated annealing algorithm used throughout this analysis. This is with the exception of discussion on a specific fitness function, which is addressed and discussed in the following section.

### 3.3.1 Solution Generation

The software architectural model provides information on each input into the test code item. This includes the range and type of the input, thus allowing the search algorithm to narrow down the search space for analysis. The initial set of inputs is pseudo-randomly chosen, using a time-seeded random function at system initialisation.

One of the principal aims for the algorithm should be to identify and then focus in on a good solution. Therefore, the derivation of a new solution on each iteration of the algorithm is based on a minor modification to the previous set of input vectors. This is introduced in Algorithm 2.

As the simulated annealing algorithm begins each iteration one input variable is randomly chosen. Depending on whether this variable can be represented as an integer, a float or a boolean; the value is either changed by 10% (integers), changed by 10 or 20% (floats) or inverted (booleans). For integers and floats the actual change performed, and whether it represents an increase or decrease, is pseudo-randomly chosen. This creates a new solution with just one variable altered. The algorithm includes a check to ensure the minimum amount integers and floats are

**ALGORITHM 2:** Solution Generation.

**Input** $CurrSolution$

**Output** $CurrSolution$

1: $I$ = random(0..$NumInputs$)
2:
3: **if** type($I$) == Integer **then**
4: $\delta = \lceil Abs(CurrSolution[I]/10) \rceil$
5: **if** $CurrSolution[I]$ - $\delta < CurrSolution[I]$.Min **then**
6: $CurrSolution[I]$ = random($CurrSolution[I] + \delta$ **or** $CurrSolution[I]$.Min)
7: **else if** $CurrSolution[I] + \delta > CurrSolution[I]$.Max **then**
8: $CurrSolution[I]$ = random($CurrSolution[I]$ - $\delta$ **or** $CurrSolution[I]$.Max)
9: **else**
10: $CurrSolution[I] = CurrSolution[I]$ + random($-\delta$ **or** $\delta$)
11: **end if**
12:
13: **else if** type($I$) == Float **then**
14: $ChangeRatio$ = Random(10.0 **or** 20.0)
15: $\delta$ = Max($MinPrecision$, $Abs(CurrSolution[I])/ChangeRatio$)
16: **if** $CurrSolution[I]$ - $\delta < CurrSolution[I]$.Min **then**
17: $CurrSolution[I]$ = random($CurrSolution[I] + \delta$ **or** $CurrSolution[I]$.Min)
18: **else if** $CurrSolution[I] + \delta > CurrSolution[I]$.Max **then**
19: $CurrSolution[I]$ = random($CurrSolution[I]$ - $\delta$ **or** $CurrSolution[I]$.Max)
20: **else**
21: $CurrSolution[I] = CurrSolution[I]$ + random($-\delta$ **or** $\delta$)
22: **end if**
23:
24: **else**
25: $CurrSolution[I]$ = NOT($CurrSolution[I]$)
26: **end if**
27:
28: return($CurrSolution$)

altered by does not tend to zero. This ensures integers and floats can traverse the zero positive/negative boundary.

If the target system model provides type ranges, as is the case with the Rolls-Royce control system, then these type ranges are taken into account when making the decision as to whether to increase or decrease the variable.

### 3.3.2 Temperature Control

The temperature (*Temp*) is the key parameter that controls the operation of the algorithm. As shown by line 5 of Algorithm 1, while solution improvements are always accepted, the Temperature weights whether or not a solution degradation is accepted or not. A higher temperature means the solution is more likely to be accepted.

The temperature is designed to slowly decrease over time; therefore ensuring that as the test progresses the algorithm becomes less accepting of worse solutions. Ultimately, the temperature decreases to such a point that the algorithm becomes a standard hill climbing algorithm.

The temperature was designed to decrease slowly from a starting point where all solutions are accepted, to a point where no worse solutions are accepted, over a period of roughly 10,000 iterations. This was shown through trial and error to provide an acceptably slow cooling period across each fitness function.

One modification from the original algorithm suggested by Kirkpatrick et al. [61] has been made. That is, if no solutions are accepted after 200 iterations, then the temperature is increased to reheat the search [62]. This reheating schedule was shown to avoid the simulated annealing algorithm being caught in a local minimum, which is regarded as one of the risks with the algorithm.

### 3.3.3 Stopping Criteria

It is generally not known when the WCET, or the worst case path, has been observed; and so the derivation of a stopping criteria is extremely difficult. Therefore, the stopping criteria used in this algorithm follows a similar approach successfully used by Tracey [63] in that it aims to execute for a sufficiently long period of time, before identifying when no further improvements are being made. The stopping criteria is ultimately balanced to allow the algorithm to execute for significantly longer than felt necessary, only stopping when no solutions have been accepted in the previous 33% of total test iterations. This is on the basis of a minimum of 1000 iterations.

This stopping criteria was defined following a process of trial and error, crucially the same criteria is used throughout the following sections to ensure a level playing field for all tests.

### 3.3.4 Derivation of a WCET

The derivation of a WCET using the search based algorithm described in this chapter follows the standard qualified Rolls-Royce process for obtaining WCET figures for software certification, as described in [13]. The process, which uses RapiTime from Rapita Systems Ltd, instruments the software under test with a series of low overhead tracepoints (called iPoints). As each iPoint is executed the processor outputs a timing trace which is captured by monitoring hardware. These timing tracepoints are input into the RapiTime tool which merges them with the source code structure in order to produce a WCET result.

The approach is qualified as a Tool Qualification Level 5 tool (Verification tool) according to DO-330 [64]. This qualification is based on the following key assumptions:

- A software architecture amenable to analysis; including defined component and hardware boundaries.

- A processing architecture that supports analysis and provides accurate non-intrusive tracing and time-stamping of software execution.

- Comparison between the code tested and the delivered code to ensure the results are representative of the final system.

The minimum requirement for the approach when targeting the Rolls-Royce processor (introduced in Chapter 2) is for the software traces to demonstrate full branch coverage, with each loop being exercised to its maximum. This requirement is derived from the tool qualification requirements for the process, as discussed further in [13].

In the context of this work, the search algorithm is used to automatically execute the software under test. As the software under test executes, iPoints (iPoints/tracepoints) are output by the processor and captured for input into the RapiTime tooling infrastructure. The aim of this work, therefore, is to produce the right coverage to allow the RapiTime tool to produce a sound WCET result.

## 3.4 Automatic Software Execution

This section focuses on the design and development of a search algorithm to automatically drive hybrid measurement-based timing analysis. The key contribution of the section is the derivation of a targeted fitness function which focuses on deriving the right data to support the analysis from across a feasible search space.

### 3.4.1 System Setup

The process followed by the WCET analysis tooling is defined in Figure 3.1. The blocks highlighted in red show manual steps, which currently form part of the formal software development process at Rolls-Royce. The blocks highlighted in blue are the additional automated steps added as part of this study. These are described below:

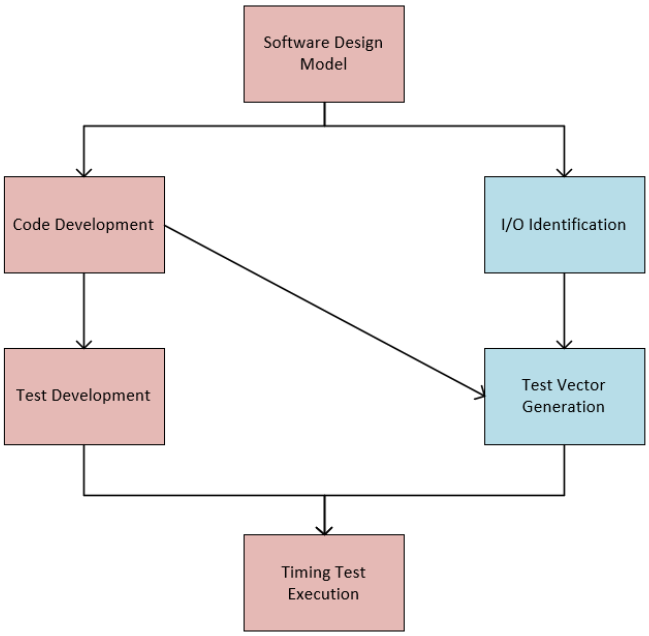


Figure 3.1: Timing Analysis Process.

- **I/O Identification** - Derivation of the inputs to each test code item from the software architectural model.
- **Test Vector Generation** - Configuration of a simulated annealing search algorithm to drive the inputs to the test code item.
- **Timing Test Execution** - Output of software timing information for input into the hybrid measurement-based timing analysis tool.

The current process, which follows the red workflow, requires engineers to define functional tests (**Test Development** stage) which provide full code coverage from the highest level of a schedulable entity, right through all sub-functions. This requirement means that timing information is not available for development engineers until a very late stage in the design lifecycle. This means engineers do not have the information they need to easily optimise their code and delays identification of timing issues until a late stage in the design lifecycle.

The ultimate aim of the process would be for the **Timing Test Execution** stage to merge results from **Test Development**, and from the **Test Vector Generation** stages for software certification. With the automated **Test Vector Generation** stage providing indicative results at an earlier stage in the software design lifecycle to the **Test Development** stage. The results in this thesis focus on the results from the **Test Vector Generation** stage.

### 3.4.2 Initial Algorithm Design

The initial search algorithm compares two different fitness functions. The first is a purely random unguided search where all solutions are accepted; this is denoted as **Ran**. The second fitness function used is the currently accepted approach as used by Wegener [19], Tracey [14] and Jones [20]; denoted **ET**. This fitness function was shown by Khan [53] to generally give the most appropiate WCET result.

ET is designed to attempt to identify the largest execution time possible. As each new solution is executed its operation is timed. The current execution time is then assessed against the previously accepted execution time. This is shown in Equation (3.1), where $CurrTime$ is a signed integer containing the time of the current solution, $PrevTime$ is the pre-

viously accepted best solution and $Fitness_{ET}$ is the fitness calculated. The subtraction of one from the time difference ensures that an identical execution time is not viewed as an improvement.

$$Fitness_{ET} = \frac{CurrTime - PrevTime - 1}{PrevTime} \tag{3.1}$$

The algorithm setup used is as defined in the previous section, with the exception of the stopping criteria for the **Ran** fitness function. The fitness function randomly accepts all solutions, and therefore measuring time since the last solution acceptance is irrelevant. As a consequence **Ran** is allowed to run for longer than any other fitness function. However, when the results are post processed, only the first $X$ are processed; where $X$ is set to the median of the other fitness functions.

### 3.4.3 Initial Results and Analysis

The simulated annealing algorithm introduced in Section 3.4.2 was applied to the test code items introduced in Section 3.2. The simulated annealing algorithm was executed fifty times, each time with a different initial seed fed into the pseudo random number generator. The fifty results from both the Ran and ET fitness functions were compared using a $\chi^2$ statistical test [65][66]. A p-value of less than 0.05 was obtained which showed that the comparison of the results were statistically significant. This provided confidence that fifty tests provided sufficient results for analysis.

The Rolls-Royce test code items used for this analysis are taken from the highest system level of the aircraft engine control system. Each task calls a number of sub-functions; the execution path followed through these sub-functions being reliant on the input arguments provided by their parent task, and so in some cases it is not possible to achieve full

block coverage of each sub-function. Therefore, the analysis of the results discussed throughout the following sections is concerned with reviewing the coverage achieved in comparison to the coverage achieved across the whole set of simulated annealing configurations.

The results presented in this section are considered to be the initial state results, based on processes for determining measurement-based timing analysis coverage defined by Wegener [19] and Tracey [14].

The initial results are summarised in Table 3.2. These show the number of tests for each fitness function[2] that managed to achieve coverage greater than 90% of the possible instrumentaton point (iPoint) coverage for each code item. The results indicated that the fitness functions obtained reliable coverage of the Mälardalen benchmarks, with the Qurt test code item receiving the lowest coverage. However, the coverage for the industrial examples was extremely poor.

In order to review the results in more detail, the coverage obtained for both fitness functions and the performance of the search algorithm was reviewed in depth for each test code item. For brevity only the analysis performed on the VCA test code item is discussed throughout the following sections. The VCA test code item was chosen because it contains a number of loops and hard to reach paths that together lead directly to longer execution times.

Figure 3.2[3] shows the iPoint coverage obtained for the VCA test code item after fifty iterations of the Ran and ET controlled simulated an-

[2]The acronyms ET_NS and Ran_NS are used to denote the two initial fitness functions. The acronym NS stands for No State and is examined further in Section 3.4.4.

[3]Each box and whisker plot throughout this thesis displays the 25th to 75th percentile as the limits of the box, with the 50th percentile (the median) marked with a bold line. The plot also shows the 5th and 95th percentiles as whiskers above and below the box, with any further outliers being shown with circle marks.

Table 3.2: The Number of Tests That Achieved Greater than 90% iPoint Coverage.

| Item | MCC | Ran_NS | ET_NS |
|---|---|---|---|
| Qsort | 21 | 50 | 50 |
| Qurt | 19 | 46 | 48 |
| Select | 20 | 50 | 50 |
| InsertSort | 5 | 50 | 50 |
| F | 154 | 50 | 50 |
| ACDF | 9 | 0 | 0 |
| ACDN | 14 | 0 | 0 |
| ACDT | 55 | 0 | 0 |
| ACDP | 27 | 0 | 0 |
| VCA | 68 | 0 | 0 |
| VCP | 94 | 0 | 0 |
| VCS | 21 | 0 | 0 |

nealing algorithms. The iPoint coverage is shown as a percentage of the total coverage possible from the top level of the function, and so offers a perceived target for each fitness function to hit.

As is shown by Figure 3.2 the ET fitness function is not too dissimilar to the Ran fitness function for the VCA test code item. Furthermore, these initial results suggest a relation between the iPoint coverage obtained and the resulting WCET calculated by the RapiTime tool. This is mirrored by the results outlined in Figure 3.3 which shows a box and whisker plot comparing the WCET obtained by each test execution.

Figure 3.4 shows the full Control Flow Graph (CFG) for the VCA test code item. The graph shows the structure of the VCA function using a mapping of the iPoints inserted into the code, each oval shaped node

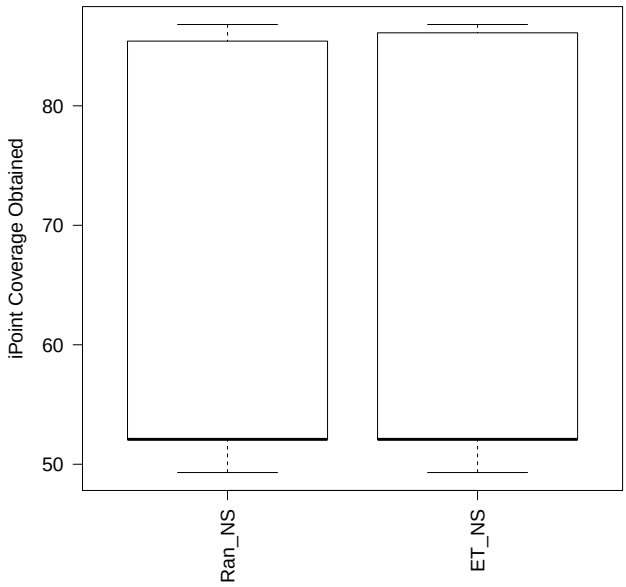


Figure 3.2: iPoint Coverage Obtained for the VCA Test Code Item.

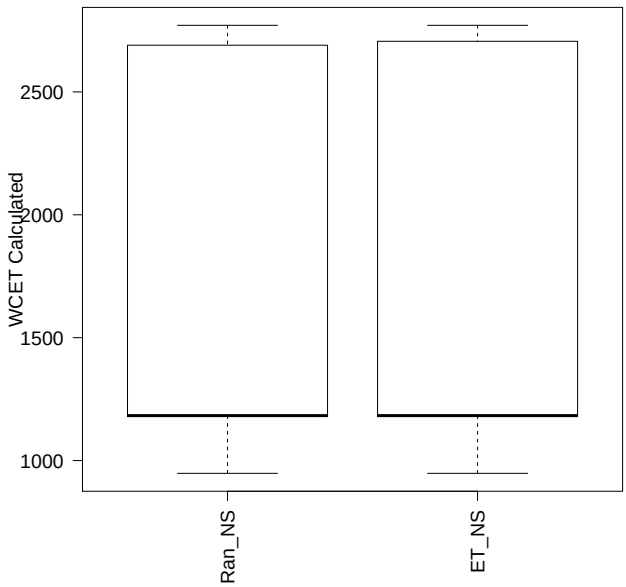


Figure 3.3: WCET Calculated for the VCA Test Code Item.

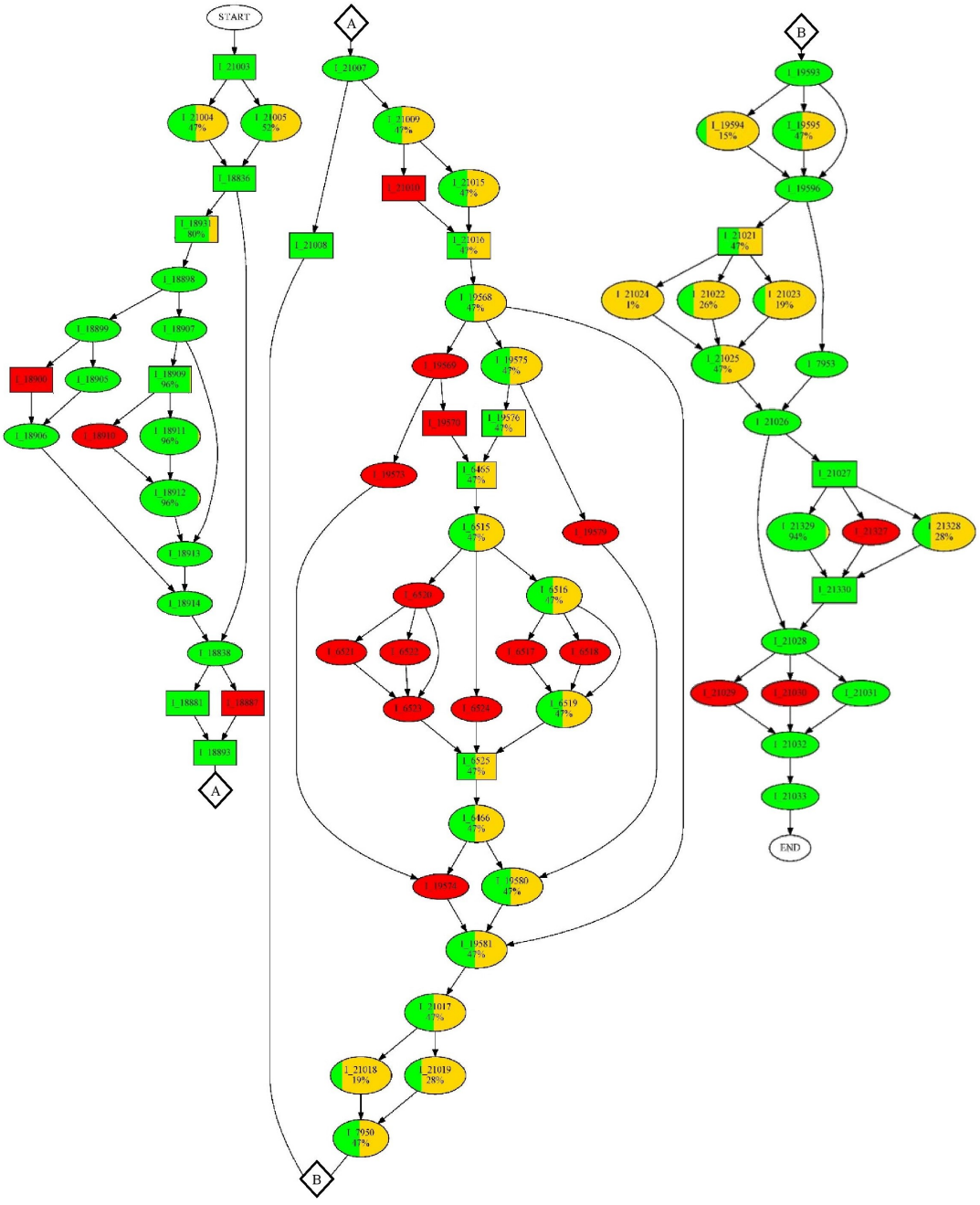


Figure 3.4: Condensed VCA Control Flow Graph.

indicates a iPoint with the arrowed-edges showing the possible routes in and out of each iPoint. This graph has been compressed as sequential non-decision iPoints have been condensed into their preceding iPoint, denoted using a rectangular node. A sequential non-decision iPoint sequence essentially refers to a sequence of iPoints that will all be executed if the first iPoint is executed, in other words there is no decision to alter the iPoint path. Each node is shaded to indicate the coverage obtained as the ET fitness function executed. A fully green node indicates that this iPoint was executed in all fifty tests, a red node indicates this iPoint was never executed. Finally, yellow/green nodes indicate a node where some, but not all, tests executed this node; with the percentage of tests and the level of green shading used to denote how many tests successfully executed this iPoint.

It is important to remember with all CFG plots shown throughout this chapter, that these graphs reflect the structure of the code, and not the execution time of the code. That is, one iPoint to iPoint transition cannot be expected to have the same execution time as another.

Inspection of these graphs illustrates why Figures 3.2 and 3.3 suggests there is a relationship between high block coverage and a sound WCET. The graph shows how one decision in particular leads to the execution of an additional 46 iPoints; adding a significant contribution to the WCET. This specific decision and path are shown in Figure 3.5, which shows that only 47% of the fifty tests traversed this path.

The code that introduces this hard to reach path is shown in Listing 3.1. The decision on line 18 represents entry into a set of fault handling code, which is executed if *condition* is evaluated false. This evaluation is based on the setting of thirteen input variables and two state variables.

Essentially this decision represents a very hard path to reach, as entry

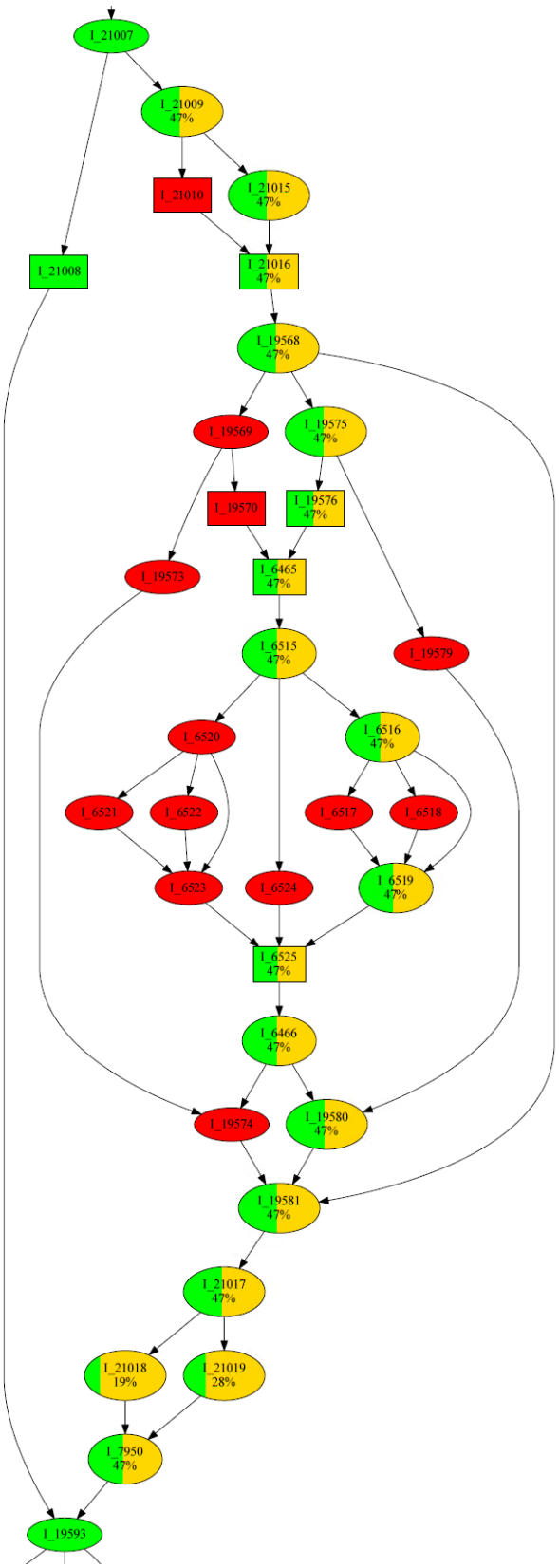


 Figure 3.5: Error Handling Code Structure Found Within the VCA Control Flow Graph.

to the path requires the correct configuration of fifteen variables. Two of these variables (*state_boolean*1 and *state_boolean*2) are controlled by previous iterations of the test, and so prove even more difficult for the search algorithm to control. This is because state variables are both inputs to the function, but also outputs. As the function is able to change their value accordingly. These values for instance may record an element of feedback in a tight control loop.

```
1   lcl_boolean1 := boolean1 or boolean2
2   lcl_boolean2 := (boolean3 and
3            (not boolean4) and lcl_boolean1)
4   lcl_boolean3 := (state_boolean1 and
5            (not boolean5 or boolean6)
6   condition := state_boolean2 or
7                boolean7 or
8                boolean8 or
9                boolean9 or
10               boolean10 or
11               boolean11 or
12               boolean12 or
13               boolean13 or
14               lcl_boolean2 or
15               lcl_boolean3
16               );
17  iPoint(21007);
18  if condition then
19     iPoint(21008);
20  else
21     iPoint(21009);
```

Listing 3.1: VCA Hard to Reach Path

Inspection of the code reveals that this path controls entry to a section of error handling code. Not only is entry into this section of code

reliant on a specific set of inputs, but furthermore these inputs, some of which indicate different data faults, actually inhibit execution of different operations elsewhere in the code. This means that in order to reach and execute this branch, a shorter execution time path must be followed to reach the branch.

In the case of the Ran fitness function, as the fitness function accepts all solutions, the algorithm is actually able to exercise a relatively large part of the search space; therefore stumbling across the long hard to reach path regularly. However, the ET fitness function is disadvantaged because in order to execute the hard to reach branch, the current solution must be allowed to 'degrade' substantially, that is, several lower execution time solutions must be accepted. This means the fitness function is essentially reliant on the random decision to accept a degraded solution and the function is not able to direct the search towards these harder to reach paths.

A similar issue was discovered on the VCP and ACDT fitness functions, and in fact can be said to be a common occurrence throughout the control system. In the case of VCA, entry to the hard to reach path is guarded by error control logic. This same logic has the effect of limiting execution of other parts of the function, i.e., one error may stop the function from performing one operation and a second may stop it from performing another operation. However, only when both are seen together does the result lead the function to take the hard to reach path. As well as error handling code, it was also found that different system parameters, such as whether the engine was idling, or whether the aircraft was on the ground, can lead to significantly different execution paths.

In addition to highlighting this hard to reach path, the CFG plots also show that the fitness functions were unable to reliably execute a large

number of iPoints. As previously noted, the execution time between two iPoints cannot be assumed to be related in any way to the execution time between a different pair of iPoints. This means that we can only have confidence in the WCET produced when we have sufficient coverage of the system.

Finally, the analysis performed so far has concentrated on code coverage, however a valid WCET also relies on a thorough exploration of the loops found within the code, which can have a significant effect on the WCET. Figure 3.6 shows a boxplot of the maximum number of loop iterations observed across each test. This is calculated as follows: the maximum number of iterations observed for each loop is recorded as each test progresses, and after completion the mean across all loops is calculated for each test and used to create this plot.

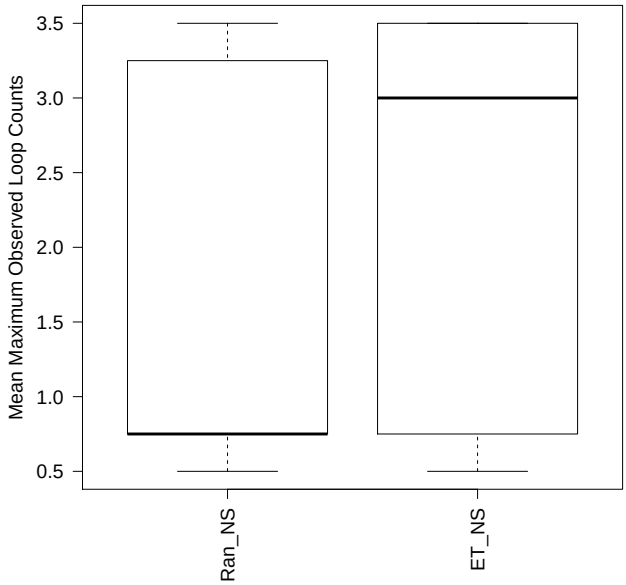


Figure 3.6: Maximum Loop Counts Observed for the VCA Test Code Item (Mean Across All Tests).

The plot shows how the ET fitness function was able to drive each loop through the VCA test code item to a significantly higher number of iterations than that of the Ran fitness function. This is understandable because higher loop counts will directly lead to higher WCET results when compared to the same execution path with a lower execution time. This suggests the ET fitness function is perhaps better suited to exploring maximum loop counts than it is higher block coverage.

This initial analysis has highlighted a number of issues with the current process as detailed below:

- The system under test contains a large amount of system state, which is carried forwards through iterations of the test. The current state of the art algorithm is not designed to, nor proves able to, handle this state appropriately to allow sufficient exploration of the system.

- Both fitness functions have difficulties reliably executing hard to reach paths.

- The use of a single fitness function focused purely on obtaining larger execution times results in poor coverage of iPoints, affecting confidence in the resulting WCET.

### 3.4.4 Assessing the Importance of System State

The analysis in the previous section highlighted that the initial current state algorithm proves unable to obtain coverage that provides sufficient confidence in a WCET estimation, and in half of cases, produces results that do not approach the WCET of the function under test. One of the issues identified was that the algorithm was not able to handle system state.

State variables are commonly found across Control System architectures, which use previous outputs for feedback to create stable control algorithms. In the initial search algorithm these state variables made it more difficult to execute hard to reach paths because the search algorithm had no direct control over them.

Two methods were investigated for handling state variables as follows. Initially, state variables were treated as other input variables and were randomly initialised and altered in exactly the same way. This improved the obtained results, however it was found that a more efficient method was to randomly choose whether to alter a state variable or to carry forward the previous iterations state variable setting. This allows the function under test to influence the state variables in a more representative way, essentially allowing the function under test to perform some of the hard work.

To analyse the results at the system level, the set of test code items were repeated fifty times. The iPoint coverage obtained, and the WCET calculated, for the VCA test code item is shown in Figures 3.7[4], 3.8 and 3.9. As the figures imply, the coverage as a whole increased significantly. This was because the search algorithm was better able to traverse state controlled decisions. Two examples of these improvements are shown in Figures 3.10 and 3.11, which compare the CFG coverage obtained during the initial analysis (on the left of each plot) to the coverage obtained by this state-handling improvement (on the right).

Secondly, the maximum number of loop iterations also increased significantly, with all variable loops seeing a notable increase in their maximum observed number of iterations.

[4]The acronym NS is used throughout this chapter to denote the initial No State results

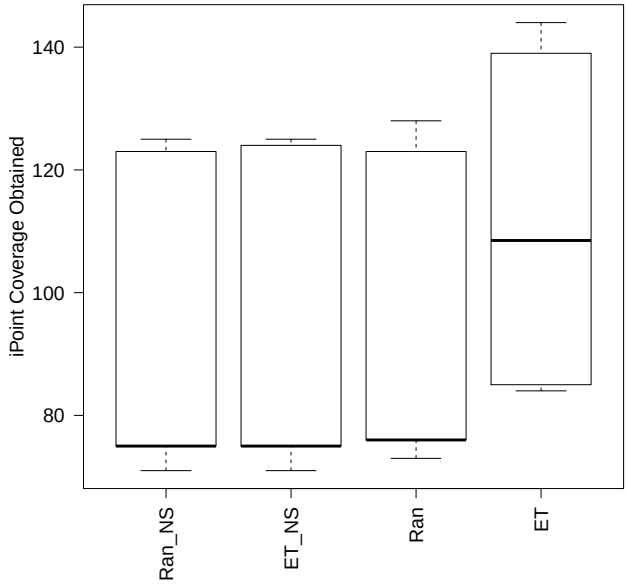


Figure 3.7: iPoint Coverage Obtained for the VCA Test Code Item, Including Addition of State Variable Control.

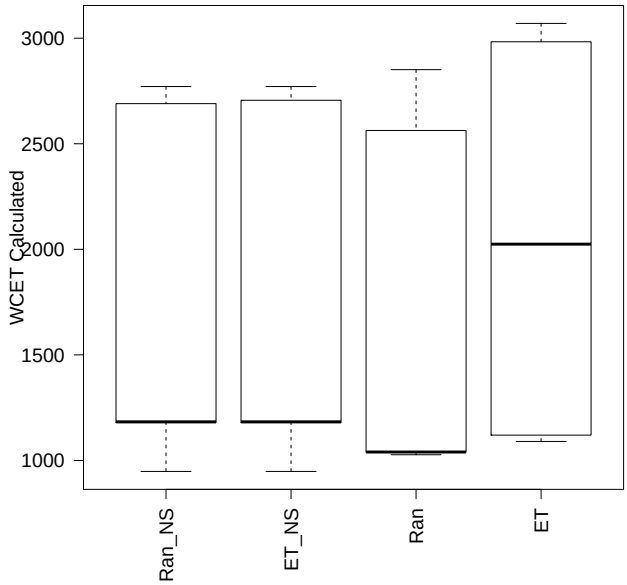


Figure 3.8: WCET Results for the VCA Test Code Item Following the Addition of State Variable Control.

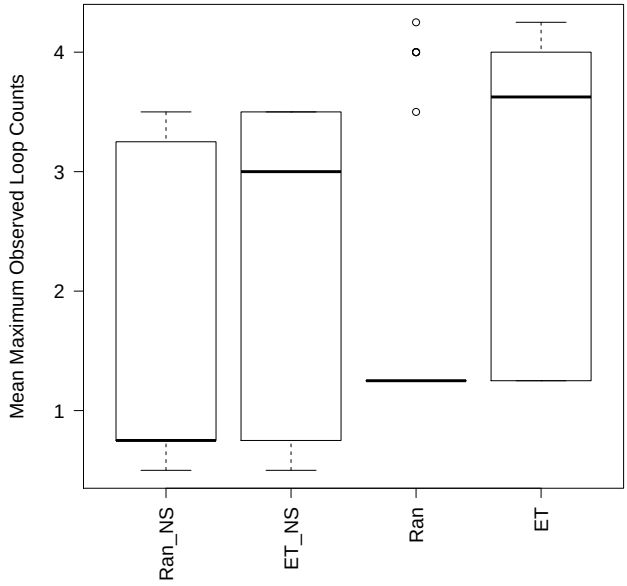


Figure 3.9: Average Loop Counts Obtained for the VCA Test Code Item Following the Addition of State Variable Control.

It is worth noting that one issue with treating state variables as inputs is that the input space that must be manipulated increases accordingly. For test targets that have a number of state variables that strongly control the execution of the function this is a necessary requirement. However, a risk is that state variables that have little or no control over the execution of the function may actually lead to a poorer exploration of the search space.

While taking control of state variables significantly increased coverage, the large range of results in the boxplots illustrate that there is still a lot of variability in the produced results. This is shown on inspection of the CFG in that the particular hard to reach path around iPoint 21007 (labelled in each CFG figure as I_21007), discussed in the previous section, is executed more often (now greater that 50% of tests), but still proves difficult to execute.

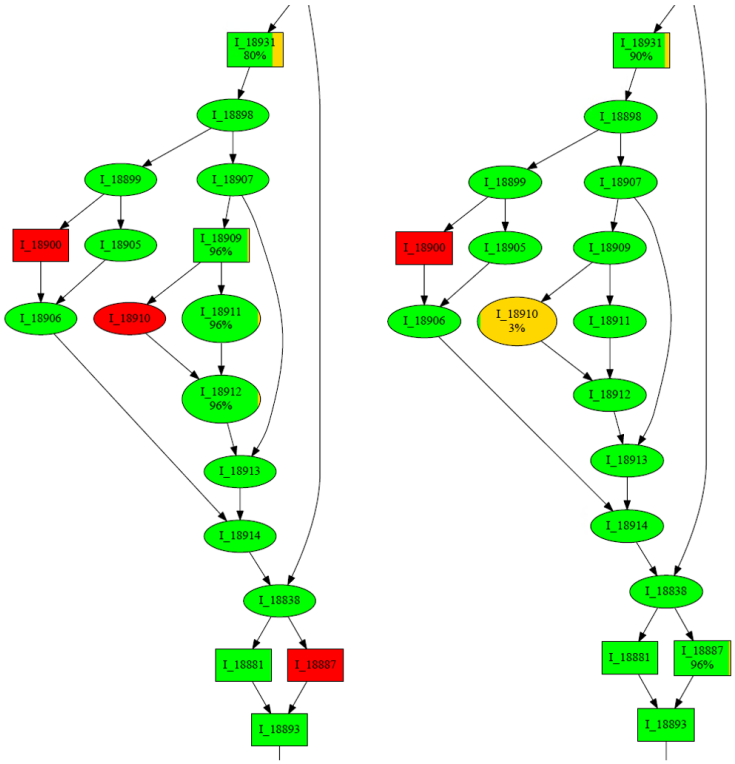


Figure 3.10: CFG Coverage Improvement Examples Following the Addition of State Variable Control.

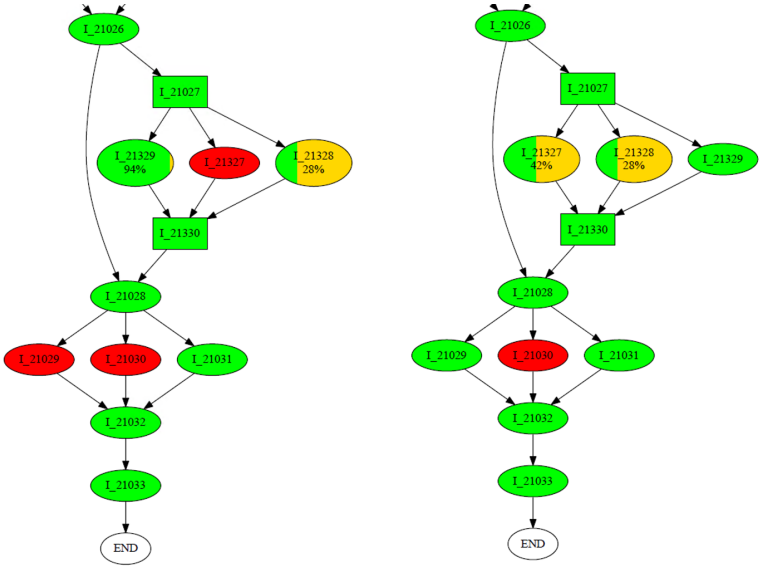


Figure 3.11: CFG Coverage Improvement Examples Following the Addition of State Variable Control.

### 3.4.5 Improving Coverage

Analysis of the results produced so far show that the search algorithm, with ET and Ran fitness functions, has so far proved unreliable at obtaining sufficient iPoint coverage across each test code item. Essentially, the algorithms do not adequately target improving poor coverage.

Analysis of the control system functions being tested have revealed a number of key properties:

- At times it may be necessary to accept significantly lower execution times, over several iterations, to identify new paths and branches.
- ET's focus on execution time results means it quickly loses focus when set against difficult to reach paths. This has been shown by the fact that a random fitness function is able to produce comparable results.
- Neither fitness functions focus on achieving full coverage. For instance they show little focus on individual blocks or branches, such as those shown in Figure 3.11. As already stated, it is not safe to assume that the number of iPoints in a block is proportional to its execution time - as a lack of coverage, leads to a lack of WCET confidence.

In order to attempt to focus on obtaining greater iPoint coverage, two new fitness functions were defined - Unique Execution Times (UET) which aims to focus on identifying new unseen paths, and Branch Coverage (BC), which aims to maximise code structural coverage.

UET aims to return a high fitness when a new path has been traversed. Paths themselves are not monitored, as maintaining a list of which paths have been executed and then checking against this list was deemed to lead to unnecessary complexity. Instead, the fitness function

keeps a record of each solution's execution time, and counts how many times each unique time has been observed. The fewer times the execution time of the current solution has been observed, the better the fitness of the solution. This is defined by Equation (3.2) where $TimeCounter$ is an array that stores a counter for each execution time value, so a newly observed execution time would return a $TimeCounter$ value of zero.

$$Fitness_{UET} = \frac{1 - TimeCounter(CurrTime)}{100} \tag{3.2}$$

The algorithm is designed as a simple path coverage metric which is designed to provide a wide execution of the solution space. As the same previously observed execution time is seen again, the fitness calculated will steadily decrease. This ensures that the space around previously observed execution times is still explored. This fitness function assumes that different paths will always have different execution times. In practise, even when executing on a deterministic processor different source code paths should lead to different object code constructs, and therefore different execution times. However, it is possible that two paths could have the same execution time, which represents a risk with this fitness function.

BC assesses the fitness at every branch through the current path. Each branch's fitness is calculated as the normalised sum of the number of edges out of the branch. The solution fitness is then calculated as the average fitness of all branches on the current path. For example, referring to Figure 3.12, if the current solution's path includes block C, (or the previously unseen blocks) B or E then the fitness calculated will be significantly higher than if the path traverses through blocks D, F, G, H or I. Thus the algorithm is weighted more towards the full execution of each branch through the code, and is weighted less towards path coverage.

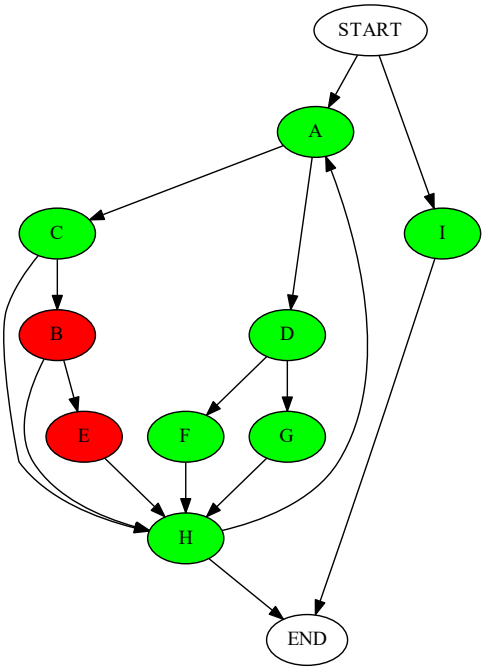


Figure 3.12: Example Control Flow Graph.

Bünte et al. [52] proposed the use of Modified Condition/Decision Coverage (MCDC) to provide WCET coverage. However, it is argued here that MCDC is not necessary in this context as it would not offer further refinement of the results over branch coverage. Ultimately this would lead to a harder search, without necessarily providing better results. For example, referring to Figure 3.12, it is not of importance how the decision was made at block D, only that both blocks E and F were executed. If the decision at D is based on a large number of variables ($N$), then the search space would increase from 2, to $2^N$.

Equations (3.3 and 3.4) show how the fitness for the current solution is calculated; where *unseen* is an array which records each edge which has not been executed, $E_b$ denotes edges from this node and $B_p$ denotes branches on the current path. The division by $B_p$ ensures the result is

normalised before being input into Line 5 of Algorithm 1. This means that the fitness is weighted more towards hitting new branches, and not against the number of branches or edges in the current path.

$$CurrFitness_{BC} = \frac{1}{B_p} \sum_{b=0}^{B_p} \left( \frac{1}{E_b} \sum_{e=0}^{E_b} unseen(e) \right) \tag{3.3}$$

$$unseen(e) = \begin{cases} 1, \text{ if e has never been traversed} \\ 0, \text{ if e has been traversed} \end{cases} \tag{3.4}$$

For the BC fitness function, as a new path is discovered the fitness will increase significantly. To balance this the fitness used by the simulated annealing algorithm is taken to be the average of the previous fifty results. A moving average is used in order to ensure that the algorithm continues to investigate newly discovered areas of the search space, by spreading out the fitness spikes seen at this point over the next set of iterations.

Figures 3.13 and 3.14 show the iPoint coverage and WCET results obtained with the new fitness functions targeting coverage.

The results indicate that the UET fitness function performed poorly compared to the latest state-controlling updates of the previous section. This was found on review to be because the search space for the UET fitness function is significantly larger, leading to a more difficult search. This is further exemplified by the fact that in order to hit some of the hardest to reach paths, the algorithm must repeatedly focus its attention on one area. However, this produces ever degrading solutions for the UET fitness function, leading the algorithm to drift away from these hard to reach paths and decisions.

The BC algorithm did not uncover new branches or iPoints that had not been executed before, but it was able to achieve greater coverage more reliably. For instance, a number of low level single iPoints were

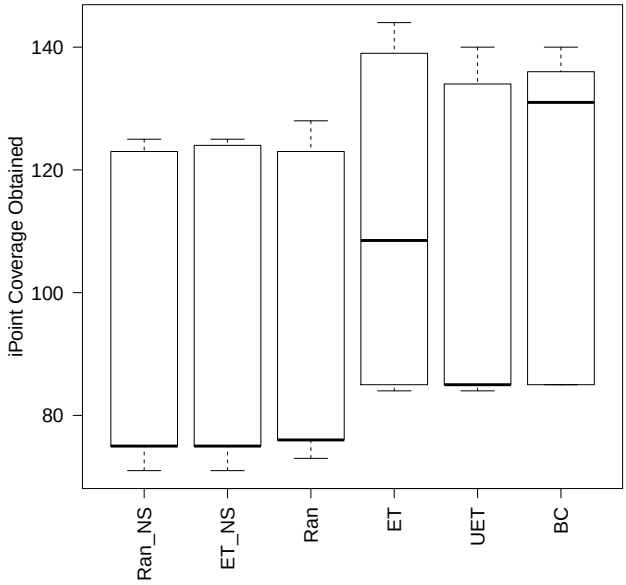


Figure 3.13: iPoint Coverage Obtained for the VCA Test Code Item Following Addition of the BC Fitness Function.

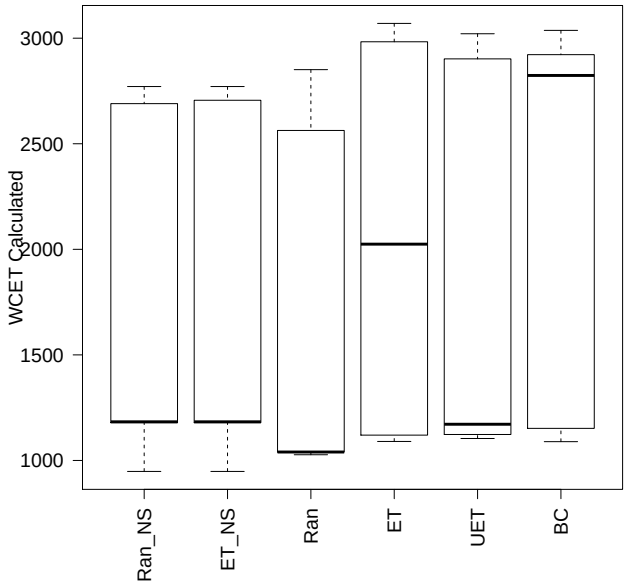


Figure 3.14: WCET Results for the VCA Test Code Item Following Addition of the BC Fitness Function.

executed more reliably. Interestingly though, the hard to reach branch introduced in Figure 3.5, while being executed in the majority of cases, was not executed in a higher percentage of tests than the ET fitness function with state control.

The results have indicated that a focus on code coverage is able to produce more reliable results than a focus on execution time, or on path coverage. However, the fitness function is still unable to reliably target hard to reach paths.

### 3.4.6 Targeting Hard to Reach Paths

The results so far show how difficult it is to reliably execute hard to reach paths. Therefore, an addition to the BC fitness function was introduced in order to focus the algorithm on executing partially-covered decisions.

Branch Coverage History (BCH) uses the same basic fitness calculation as BC (Equation 3.3). However, as each branch through the current solution's path is analysed, the input vector used to drive the current solution is stored against that branch. If after fifty iterations the solution has been rejected continuously, then the set of outgoing edges that have not been fully executed is examined, and one is chosen at random. The input vector stored against this branch is then adopted as the new input vector. This is designed to attempt to lift the algorithm from poor solutions and focus it on the area around branches that have only been partially executed.

$$Solution\_Array[b] = CurrSolution, b = 0..B_p \tag{3.5}$$

$$NewSolution = \begin{cases} \text{GenNewSolution}(CurrSolution) \\ \qquad \text{if } Reject <= 50 \\ Solution\_Array[\text{rand}(B_{NFE})] \\ \qquad \text{if } Reject > 50 \end{cases} \tag{3.6}$$

Equations (3.5) and (3.6) describe how the algorithm operates. On each iteration the current solution ($CurrSolution$) is recorded against each branch found upon the current path, as denoted by $B_p$. Equation (3.6) replaces line 3 of Algorithm 1. On each iteration if the previous fifty solutions have been rejected then the next solution ($NewSolution$) is set to equal a solution taken from the $Solution\_Array$. The array value chosen is selected from the set of solutions that drive branches that have not been fully executed ($B_{NFE}$).

Figures 3.15 and 3.16 update the VCA results with the new fitness function. The coverage obtained overall matched BC, however the number of tests which executed the hard to reach path increased to 65%, compared to the original 50% for the ET fitness function.

### 3.4.7 Increasing Confidence

The focus of the fitness functions so far has been on improving branch coverage. However, the second element that must be considered is the number of iterations of each loop through the system. If loops are not executed to their maximum number of iterations; then this may have an optimistic effect on the resultant WCET. Therefore, an expansion towards achieving the maximum number of iterations of each loop was suggested.

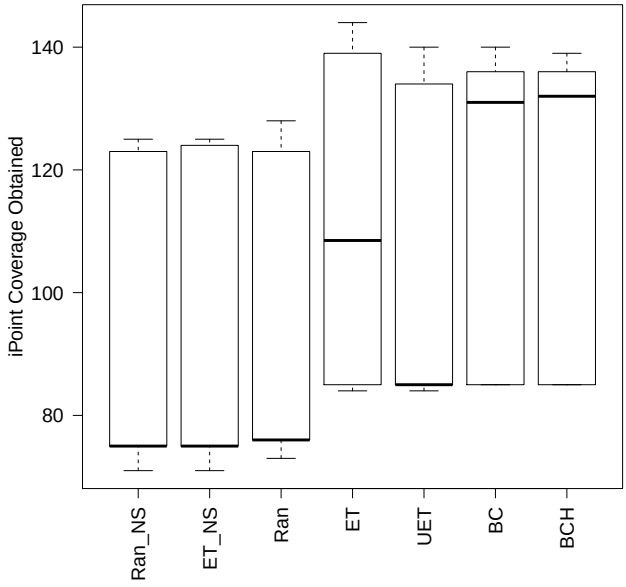


Figure 3.15: iPoint Coverage Obtained for the VCA Test Code Item Following Addition of the BCH Fitness Function.

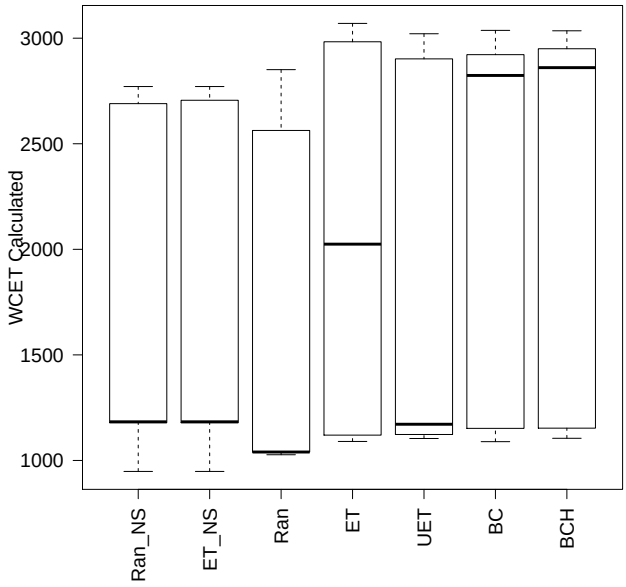


Figure 3.16: WCET for the VCA Test Code Item Following Addition of the BCH Fitness Function.

Loops (Lo) calculates the average number of iterations of each loop on the current path. The result is then normalised using the maximum observed number of iterations. The algorithm is based on previous work by Khan [53]. Using the CFG in Figure 3.12; block H will be identified as a loop back edge, the fitness for the solution in this case will be calculated as the number of times block H has executed on the current path. In cases where there is more than one loop then the average number of iterations for all loops in the test item will be calculated as the fitness. As a final step the fitness is normalised by dividing the fitness by the highest fitness ever observed. Equation (3.7) shows the operation of the fitness function, where $L_P$ represents the number of iterations for each loop on the current path, and $N_L$ the number of loops on the current path.

$$CurrFitness_{Lo} = \frac{1}{Fitness^{max}} \frac{1}{L_p} \sum_{l=0}^{L_p} (LoopIter(l)) \tag{3.7}$$

Finally, Branch Coverage Loops (BCHLr) combines a search for high branch coverage, with one for high loop counts. The function combines the result produced using BCH, with the result using Lo to produce a fitness function that begins by trying to identify unseen blocks, but evolves as the search progresses to concentrate on identifying higher loop counts. Equation (3.8) illustrates how the fitness is calculated. $W_L$ is used to weight the effect of the loop fitness calculation (Lo) and is initialised to zero.

$$CurrFitness_{BCHLr} = \frac{(W_L * CurrFitness_{Lo}) + CurrFitness_{BCH}}{1 + W_L} \tag{3.8}$$

As the test progresses, and the branch coverage obtained increases, then $W_L$, the loop fitness weighting, is increased. This changes the priority

of the fitness function as the test progresses from initially focusing on branch coverage, towards a focus on maximising loop counts.

Figure 3.17 shows the iPoint coverage obtained for the additional fitness functions. As can be noted, the iPoint coverage obtained for the Lo function is on a par with the coverage obtained for the branch coverage fitness functions, including the fitness function with history. It can also be observed that the BCHLr fitness function was able to outperform the branch coverage fitness functions, even though the addition of loop counts into the fitness function equation does not have an impact on branch coverage fitness. The reason for this increase was caused by the design of the VCA test code item being studied. The function includes a number of loops, including one around the hard to reach path which proved elusive to other fitness functions. This means the BCHLr and Lo are unfairly weighted towards this path. This anomaly is addressed in the following section which discusses a wider fitness function evaluation.

Figure 3.18 shows a box and whisker plot reviewing the maximum loop counts observed across the set of tests for the VCA test code item. The results indicate that the fitness functions focusing on loop counts were able to obtain the highest average number of iterations for each loop, with the branch coverage and ET fitness functions closely behind. As shown by Figure 3.19 the iPoint coverage and loop counts observed are mirrored by the WCET results. The figure shows the fitness functions taking into account loop counts, and crucially the combination of branch coverage and loop counts, as producing the best results.

This section has discussed the design of a series of fitness functions aiming to obtain sound and reliable measurements to support timing analysis. It has presented a logical design flow based on the properties of the code under test. The next section now explores how these fitness function perform against a wider evaluation.

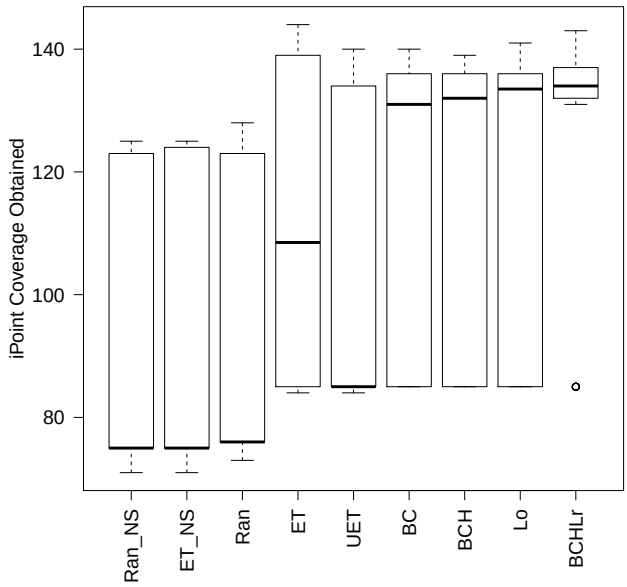


Figure 3.17: iPoint Coverage Obtained for the VCA Test Code Item Following Addition of the Lo and BCHLr Fitness Functions.

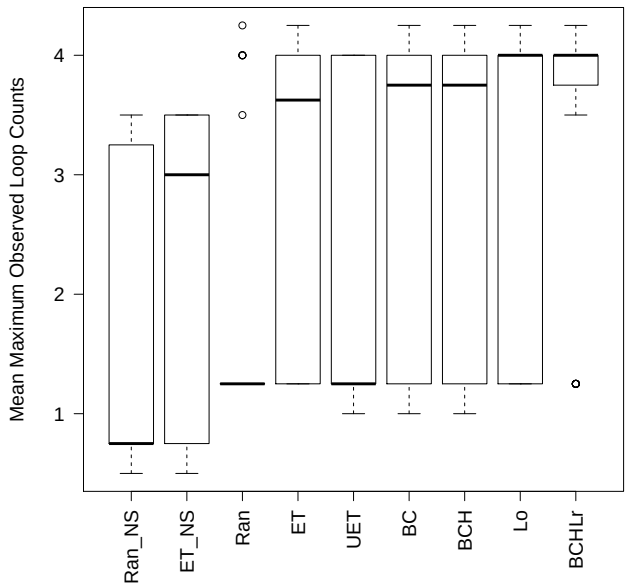


Figure 3.18: Maximum Loop Iterations Observed for the VCA Test Code Item Following Addition of the Lo and BCHLr Fitness Functions.

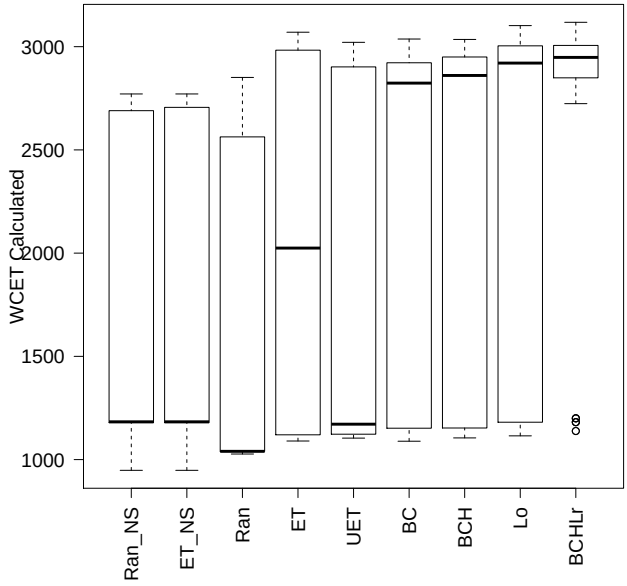


Figure 3.19: Average Maximum Loop Counts Observed for the VCA Test Code Item Following Addition of the Lo and BCHLr Fitness Functions.

### 3.4.8 Fitness Function Evaluation

In order to assess the effectiveness of the set of developed fitness functions; each algorithm was executed against the set of test code items introduced in Section 3.2 fifty times. In all cases each fitness function was executed for significantly longer than perceived necessary, as introduced by Section 3.3.3. Crucially, all tests across all fitness functions and code items used the same search algorithm configuration, with only the fitness function changing.

All results analysed throughout this section were produced from tests which are able to control state variables, as introduced by Section 3.4.4.

It has already been stated that an industrial WCET tool should be expected to provide sound results efficiently that can be produced and reproduced as part of an affordable process that ultimately provides a

system designer with confidence in the result. The fitness functions developed as part of this section are now assessed against this criteria based on three key areas - efficiency, confidence and sound WCETs.

**Efficiency**

A typical Rolls-Royce Aircraft Engine Control System consists of several thousand functions that must all be evaluated to identify the WCET of the system as a whole. Therefore, even though it could be argued that a simple search algorithm left to execute indefinitely could eventually stumble upon perfect results, in an industrial setting this approach is not practical or cost effective. An industrial grade WCET must be trusted to analyse results efficiently and reliably.

To assess the efficiency of each fitness function, the HWM for each test iteration was collected during execution. The mean HWM for each fitness function at each iteration was then calculated across all fifty test runs, and plotted for analysis. For the majority of the test code items the test results for each fitness function varied by less than 10% as each test progressed. However, in the cases of ACDT, VCP and VCA the difference was more profound. The results for ACDT and VCA are shown in Figures 3.20 and 3.21.

Firstly, all individual tests for every fitness function on all test code items completed in less than 20,000 trial iterations. This took approximately twelve hours to execute. In the case of the simple test code items each trial completed in approximately 2000 iterations, which took on average one hour to execute.

In an industrial context if each trial takes one hour, provided there is enough server power to allow multiple concurrent tests, this could be deemed acceptable. However, for the more complex functions the fact that each individual trial takes twelve hours illustrates the importance

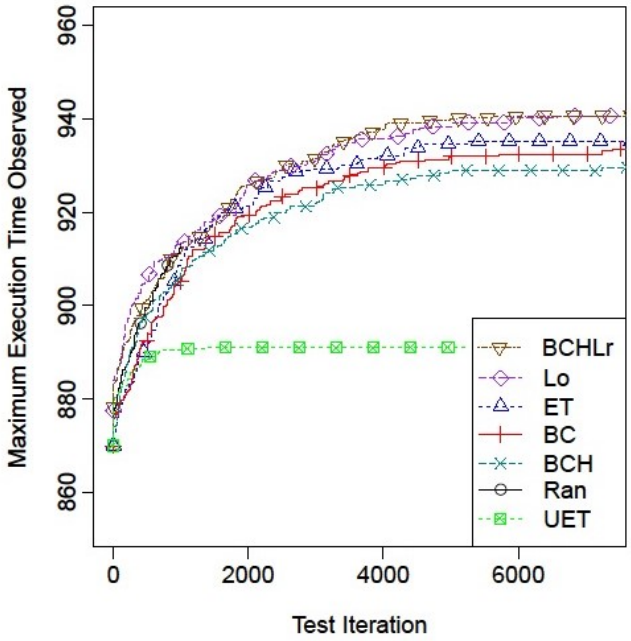


Figure 3.20: ACDT Mean HWM Observed as the Test Progresses.

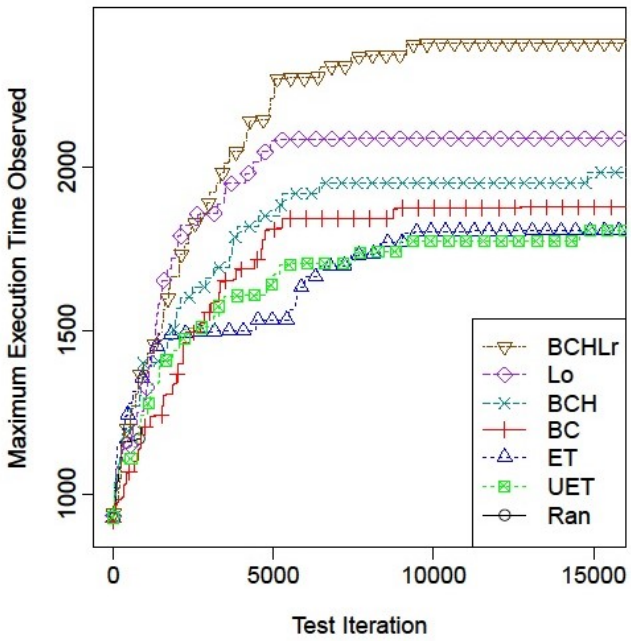


Figure 3.21: VCA Mean HWM Observed as the Test Progresses.

of identifying a test result efficiently. It also illustrates the importance of the algorithm identifying when no more progress is being made, and to stop searching. This is particularly pertinent for the small functions which may not see a significant improvement across their test run.

Figure 3.20 shows the mean HWM for the ACDT test item over time, for each fitness function, which provides a representation of test progression. The graph shows how as each test progresses all the fitness functions were able to obtain results similar to each other, with the exception of UET. One possible reason for this is the size of the input space and number of complex paths through this function, which the UET fitness function was not able to manipulate as effectively.

VCA, shown in Figure 3.21, on the other hand presented a much larger difference in mean HWM figures. In this case BCHLr was able to produce the best observed HWMs throughout the test. By 10,000 iterations all the fitness functions had stopped improving.

Both Figures 3.20 and 3.21 illustrate the difficulty of identifying an appropriate stopping criterion. The same criteria, as introduced by Section 3.3.3 was used for all fitness functions to ensure a fair test, however in the case of the Ran and UET fitness functions because the tests quickly cease to improve on their results the fitness functions quickly stop searching. Arguably, the Ran and UET fitness functions could use a different stopping criteria to force them to execute for longer, however, this approach was not taken during this analysis due to a desire to avoid giving one fitness function a further advantage over another.

In summary, the progression of each fitness function's progression over time illustrated that all the algorithms were capable of producing results efficiently for the simple code functions. For the more complex functions BCHLr performed well over all functions; with Lo, ET and BCH able to

produce good results in most of the test code items.

**Confidence**

Industry cannot rely on just reliably achieving a high predicted WCET because for certification it is important we are able to argue about confidence in the degree of system coverage. It is not appropriate, or possible, to argue that the WCET is understood without illustrating sufficient coverage has been obtained. The objective of this section is to evaluate the relative branch coverage and loop iteration counts achieved by the different approaches by reviewing the iPoint coverage during each test.

Table 3.3 shows the number of test runs for each fitness function that obtained iPoint coverage within 90% of the maximum possible.

Table 3.3: Objective 2 - The Number of Tests That Achieved Greater than 90% iPoint Coverage.

| Item | MCC | Ran | ET | BC | BCH | Lo | BCHLr | UET |
|---|---|---|---|---|---|---|---|---|
| Qsort | 21 | 50 | 50 | 50 | 50 | 50 | 50 | 50 |
| Qurt | 19 | 46 | 48 | 49 | 43 | 48 | 48 | 49 |
| Select | 20 | 50 | 50 | 50 | 50 | 50 | 50 | 50 |
| InsertSort | 5 | 50 | 50 | 50 | 50 | 50 | 50 | 50 |
| F | 154 | 50 | 50 | 50 | 50 | 50 | 50 | 50 |
| ACDF | 9 | 47 | 48 | 50 | 49 | 49 | 50 | 38 |
| ACDN | 14 | 50 | 48 | 45 | 46 | 48 | 49 | 34 |
| ACDT | 55 | 50 | 50 | 49 | 50 | 50 | 50 | 42 |
| ACDP | 27 | 19 | 25 | 45 | 43 | 40 | 45 | 14 |
| VCA | 68 | 17 | 25 | 28 | 30 | 32 | 42 | 24 |
| VCP | 94 | 10 | 25 | 28 | 25 | 32 | 35 | 30 |
| VCS | 21 | 50 | 50 | 50 | 50 | 50 | 50 | 50 |
| Mean | | 40 | 43 | 45 | 45 | 46 | 47 | 40 |

For all of the simpler test code items (those with McCabe complexity of 21 or less) the iPoint coverage for all fitness functions was 100% in most cases. The other tests showed lower iPoint coverage for some of the fitness functions. Again this indicated for simple code items all of the fitness functions were able to obtain reliable results.

For the more complex functions the variance between fitness functions was more profound. A number of the functions, such as ACDP and VCP, contain a number of hard to reach paths. For ACDP for instance the branch coverage fitness functions were able to narrow in on these paths more reliably, and thus achieved higher iPoint coverage.

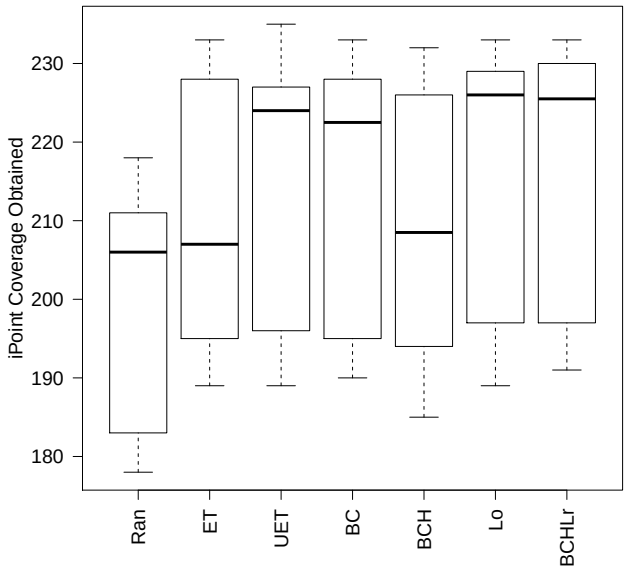


Figure 3.22: iPoint Coverage Obtained for the VCP Code Item.

In the case of VCP, illustrated in Figure 3.22; again ET and BC failed to obtain consistent branch coverage. One contributing factor to this was the size of the input space for VCP, which is considerably larger than a number of the other test code items and results in a much larger search space.

The second requirement for providing confidence in the produced WCET results is sufficient exploration of loops; the aim being to maximise loop iteration counts. Figure 3.23 shows the mean maximum loop counts obtained for each test run of the VCP test code item. The BCHLr and Lo fitness functions were able to obtain the highest median loop counts; however there were a significant number of outliers sitting high above the median, with UET obtaining the highest single outlier. As with the coverage results presented in Figure 3.22 this was attributed to the size of the VCP search space, making the search difficult for all fitness functions to obtain coverage.

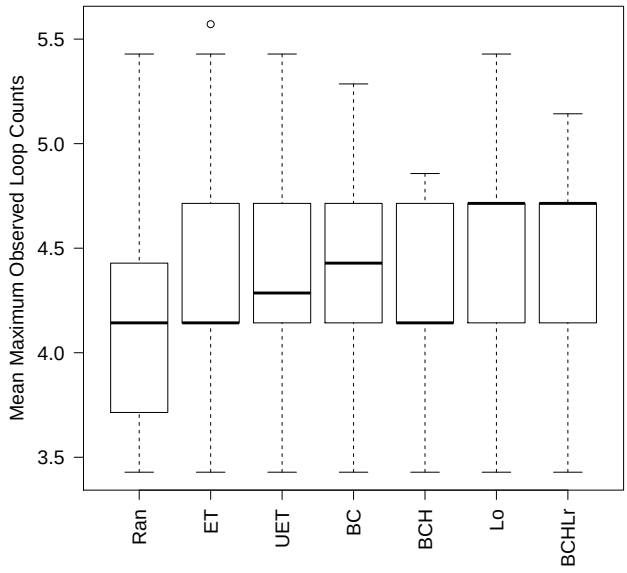


Figure 3.23: Maximum Loop Counts Obtained for the VCP Code Item.

Finally, Figure 3.24 shows the maximum loop counts obtained for the Insert Sort test code item. The results indicated the highest range in maximum loop counts observed across all test code items, which was attributed to the fact the source code item contains a nested loop structure, the execution of which is wholly reliant on the input test vectors. All fit-

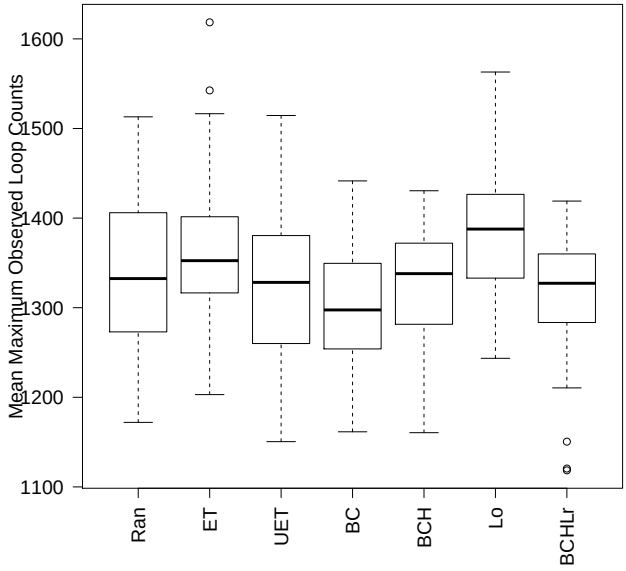


Figure 3.24: Maximum Loop Counts Obtained for the Insert Sort Code Item.

ness functions instantly obtained full block coverage through this fitness function, and so the ET and Lo fitness functions, whose focus included increasing loop counts straight away, were then better able to focus on increasing the double nested loop's number of iterations. The BCH and BCHLr's searches however were affected because the fitness function's initial focus on obtaining large block counts had no purpose and saw no improvement, which appeared to negatively affect the search.

In summary, BCHLr has again been shown to provide reliable results across all test code items. Other fitness functions, such as BCH or Lo, were able to obtain similar results for some test code items, but also produced poorer results in other test code items, such as VCA and VCP. This was shown to be because BCHLr was able to execute specific hard to reach paths. Without a focus on reaching these paths; other fitness functions like ET were unable to reliably achieve high iPoint coverage.

### Sound WCETs

The final objective to assess the set of fitness functions against is in the production of sound WCET figures. This objective is analysed by reviewing the results produced by the RapiTime tool after parsing each test's measurements. As the Actual-WCET is not known each individual test was executed for significantly longer than necessary, with the results from all tests being compared against each other. This allows an assessment to be performed into the reliability of each individual fitness function, with particular attention paid to the results when compared to the ET fitness function. A comparison between the maximum HWM obtained and the WCET calculated is performed to assess how the data input guides the result. The median is used throughout this evaluation as it best reflects where the majority of the results lie. This follows the aim of this analysis - to produce good approximations the majority of the time, rather than a better result only once.

Finally, a statistical analysis was used to assess whether any of the WCET distributions from each fitness function was significantly different from any other. This was used in order to confirm the results represented a large enough sample to show significance [65]. The data analysed is non-parametric (does not follow a normal distribution) and only one data source was used therefore a Friedman test with an alpha level of 0.05 was chosen for the analysis. This revealed that there was a significant difference between the fitness functions for all tests, this is denoted in this section as the Friedman chi-squared result ($\chi_r^2$), the degrees freedom and the p value. Following the Friedman test a Wilcoxon-Nemenyi-McDonald-Thompson [66] was used to reveal which fitness functions produced significantly different distributions.

For the smaller code items, with McCabe complexities of 21 or less,

the variance between each fitness function was very low. All fitness functions obtained WCET figures within 10% of each other, with ET generally performing best. For the InsertSort test code item the overall Friedman test result was $\chi^2_r(6) = 67.8$, $p < 0.01$[5] which indicated an overall significance. Figure 3.25 illustrates the results of the Wilcoxon-Nemenyi-McDonald-Thompson test; each fitness function was compared against each of the other fitness functions. The bar for each result indicates which fitness function in the comparison performed best by showing the result of total difference between each fitness function. A shaded boxplot indicates a significant result (p <0.05). For instance, the first bar on the left indicates that BC performed slightly better than BCH, as the result of 'subtracting' BC from BCH provides a negative result. However, the result is not significant and so should be disregarded.

The figure shows how the Lo, ET and BCHLr fitness functions were able to achieve consistently better results than the fitness functions that just focused on iPoint coverage alone. This was most likely because the InsertSort test code item only contains five iPoints, which are fully executed very early in the search, and so the branch coverage functions quickly moved to an unguided search.

For the Mälardalen WCET benchmark functions in general there was a large difference between the HWM and estimated WCET. Qsort for instance observed a HWM for each fitness function of between 8% and 15% of the calculated WCET. In a similar vein the difference between fitness functions was marginal for the Select code item; however the HWM observed was less than 1% of the WCET calculated. This was because of the effect of an infeasible path which spans over a triple depth nested loop.

[5] $\chi^2$ results throughout this thesis are denoted using the following terminology - $\chi^2$([*degrees of freedom*], n = [*number of samples*)] = [*result*]), [*statistical significance*].

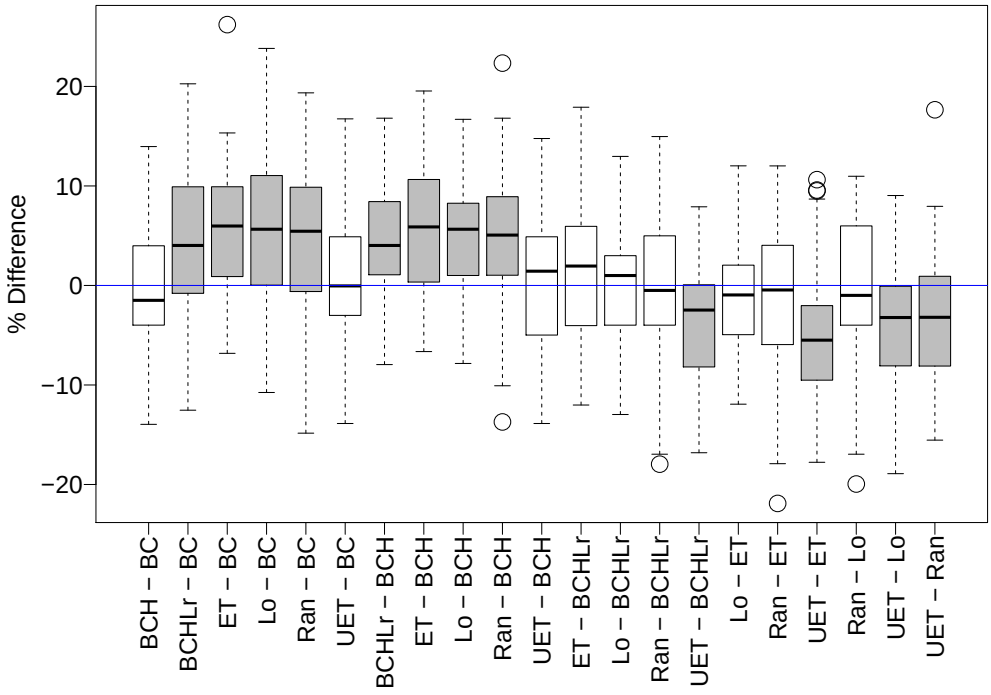


Figure 3.25: Comparison of the distribution differences for each fitness function combination, for InsertSort. Shaded bars indicate statistically significant results.

These results indicate the benefit of building a hybrid measurement-based tool on top of an automatic software execution mechanism rather than just relying on the latter to produce reliable results.

The largest difference between the different fitness function's WCET results was produced by the VCA, ACDP and VCP code items.

The VCP code item, shown in Figures 3.27 and 3.28, exhibited a significant variance of up to 30% between WCET figures ($\chi^2_r(6) = 84.9, p < 0.01$). This was found to be due to the size of the input space which led to a significantly larger search space and a lower resultant iPoint coverage. As well as this the function contains a number of loops whose execution is reliant on the data input into the test code. This highlighted

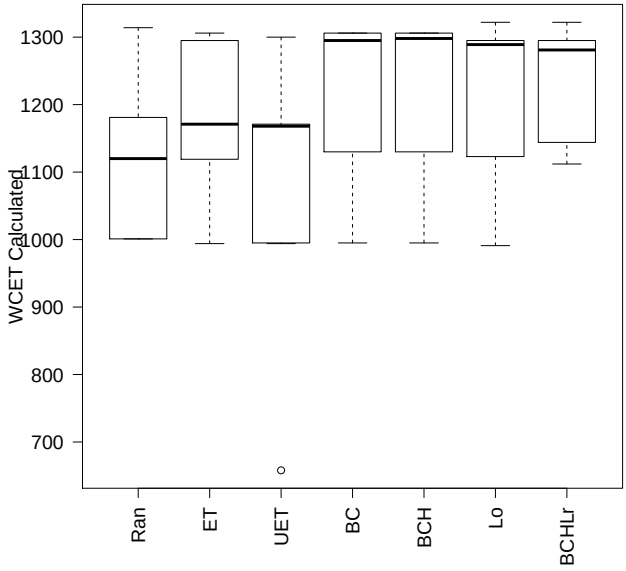


Figure 3.26: WCET Results Calculated for the ACDP Test Item.

one flaw with the BCH/BC fitness functions, in that they were unable to focus the algorithm on increasing the number of iterations of the loops found in the test code. The loop coverage fitness functions, in particular BCHLr, were able to exploit this type of code construct, and produced the highest, most consistent result.

The VCA function, as discussed earlier, showed a strong correlation between iPoint coverage and estimated WCET. For VCA the variance between the maximum and minimum estimated WCET results approached 50%, as shown by Figure 3.29 ($\chi^2_r(6) = 91.5, p < 0.01$).

## 3.5 Summary

This chapter studied how reliable, automated, timing analysis can be performed in order to identify the timing properties of a system. The existing approaches and literature, even when applied to deterministic

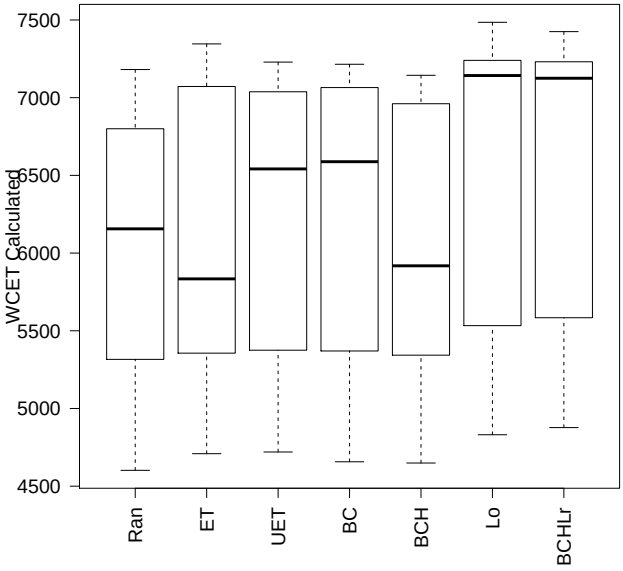


Figure 3.27: WCET Results Calculated for the VCP Test Item.

processors, is lacking when it comes to the definition of approaches to generate data to support measurement-based timing analysis, with most techniques relying on data already being available. The chapter has aimed to identify the applicability of optimisation algorithms to the generation of such data.

The chapter identified the requirements for the generation of timing data to support measurement-based timing analysis, before identifying appropriate techniques for obtaining the right data, reliably. The approaches were assessed incorporating both open source benchmarks, and closed source real industrial examples, to analyse their abilities.

The ultimate achievement of this chapter has been to produce a set of fitness functions aimed to drive the efficient, and automated, definition of task timing properties. The results provide indications of maximum loop counts as well as HWMs and WCETs for each task. Furthermore, the results provide a comprehensive set of task timing profiles from execution

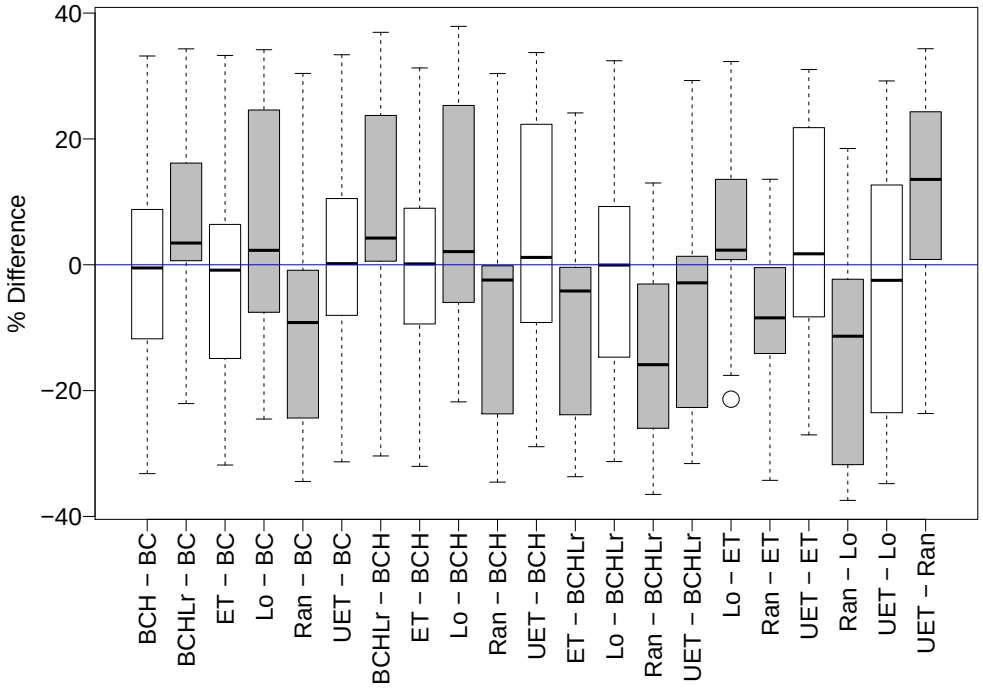


Figure 3.28: Comparison of the distribution differences for each fitness function combination, for VCP. Shaded bars indicate statistically significant results.

with representative inputs, these profiles provide indications for how each task may execute in operation.

Since the work performed in this chapter was published in [37], Lesage [67] has demonstrated that the approach can be scaled to a full aircraft engine control system. Given the scale of the analysis performed by Lesage, the work focused on the BCHLr, Ran and ET fitness functions only. The work, which included an extensive amount of tooling infrastructure development, targeted a live project that was certifying a sister system to the control system introduced in Chapter 2.

One area not examined during this Chapter, and is left to future work, is an evaluation to identify whether combinations of the developed fitness functions could prove even more effective. The BCHLr fitness function

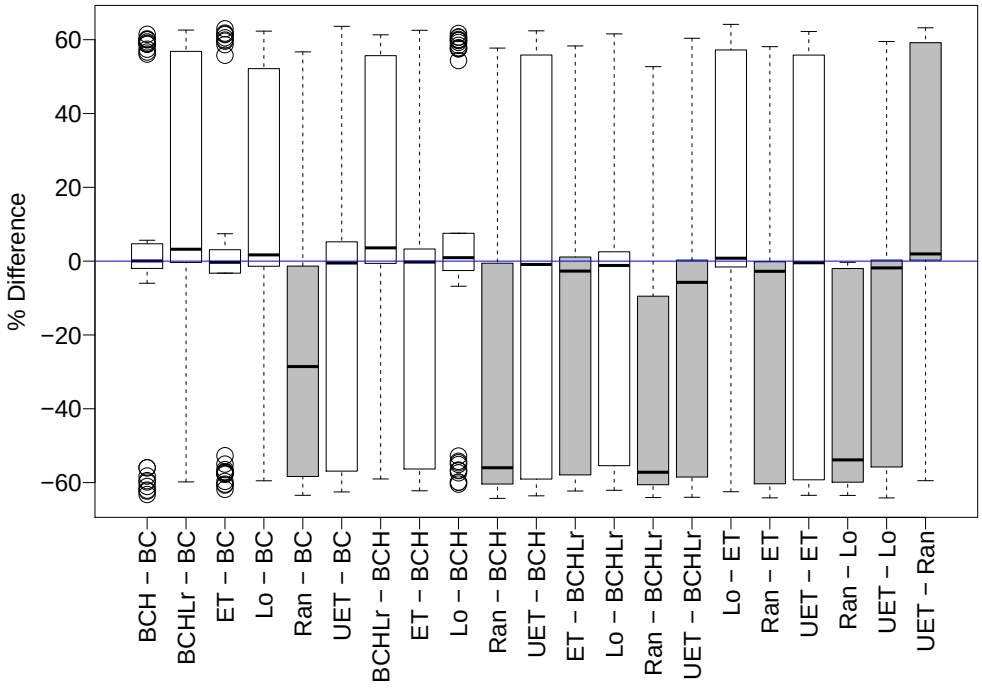


Figure 3.29: Comparison of the distribution differences for each fitness function combination, for VCA.

for instance is a combination of branch coverage and loop counts. Future work could examine whether the fitness function would also benefit from the addition of execution time measurements, or even a measure of more advance processor feature consequences, such as cache misses.

Chapter 4 now steps away from WCET analysis to discuss the development of a mixed criticality system. The system proof of which relies on the timing information provided in this chapter.

# Chapter 4

# Developing Mixed Criticality Systems for Real Platforms

Real time embedded software tasks developed for safety critical systems, such as civil aircraft engine controls, are typically developed according to a specific Development Assurance Level (DAL) [6]. The DAL indicates a criticality level for a component and is assigned based on the consequence to the system's safety that a failure of this component could cause. This thesis considers the model presented in DO-178C [6] with the criticality level of the task denoted by $L_i$ where $L_i \in \{A, B, C, D, E\}$. Level A indicates the highest level, E indicates the lowest level such that $A \geq B \geq C \geq D \geq E$. It is typically assumed that the amount of effort assigned to producing enough evidence to prove the correct operation of a software component monotonically increases with its DAL [24][1].

In accordance with DO-178C [6], where two components developed against different DALs are integrated in the same system; it is neces-

[1] Although Vestal [24] cited DO-178B, and therefore only contained criticality levels A, B, C and D, the work in this thesis assumes a simple extension to incorporate DAL E

sary to guarantee that high criticality components are protected from 'unproven' low criticality components. In other words, as defined by Rushby [68]; partitioning should be implemented with the aim of ensuring the behaviour and performance of software in one partition is unaffected by the software in other partitions. This is required because the lower DAL component does not have the same level of evidence of lack of error as the higher DAL component, and so partitioning must be used to isolate higher DAL components from any failure in lower DAL components. There are two forms of partitioning that must be employed [68]: temporal partitioning, which is concerned with the response time of a component; and spatial partitioning, which is concerned with the hardware and memory space of a component.

From an industrial point of view; approaches in the past have required all software components that execute on a processor be verified to the same DAL. This means a system that employs software of multiple criticality levels maintains physical partitioning across multiple processors to separate different software components with differing DALs. Alternatively, where system requirements make it desirable for lower-criticality software to share the same processor as a high-criticality piece of software, then they are both developed to the same highest criticality level. This approach leads to higher development and production costs, as well as increased overall size, weight, and power use.

A further advantage to the use of Mixed Criticality System (MCS) schedulers is in their ability to help system developers cope with WCET pessimism. It is known that WCET analysis techniques induce significant pessimism. From reviewing the current system introduced in Chapter 2 it was identified that there are three principal reasons for this. Firstly, the use of defensive programming which introduces error handling logic;

secondly through the introduction of infeasible worst case paths; and finally through the use of pessimistic loop bounds.

Arguably the WCET and associated schedulability analysis should incorporate error handling logic, with its addition not being considered pessimistic. However, the logic may indicate entry into a reactive system state, such as system shutdown or system reset, which means that incorporation of this handling logic into the schedulability analysis of the system could be argued as pessimistic. Whereas infeasible paths are frequently inadvertently introduced into complex control systems as software components are designed to handle different system level events. For example, an aircraft engine control system cannot be expected to respond to an engine overspeed on the same cycle that it responds to an engine underspeed. However, the system level schedulability analysis may assume this to be the case. Finally, pessimistic loop bounds are induced when the control system is viewed at the system level. As, for instance, communication interface WCET analysis assumes messages are sent at the highest possible rate, when in practise actual transmission rates nay be significantly lower.

When viewed at the macro level it is easy to identify that such analysis consequences are pessimistic. However, when viewed at the system level encompassing the analysis of thousands of software functions, identifying and combating such pessimism quickly becomes infeasible in a cost effective timeframe.

One way that forms of pessimism can be identified at the system level is by comparing analysed WCETs to system test observed HWMs. A safety critical system can be expected to be subjected to an exhaustive test campaign, which should test the system extensively. This provides a test setup well suited to providing tight, but potentially optimistic, HWM

times which can be used in comparison to estimated WCET times to identify areas of particular pessimism. The generation of such metrics, for example, was explored in Chapter 3. However, the risk with attempting to use HWMs for analysis improvement, or to reduce pessimism across a system, is that the times may introduce optimism - therefore their use must be carefully controlled with necessary system protections put in place. One of the focuses of this chapter is on the development of such controls.

This chapter explores two avenues. The first is how an appropriate mixed criticality partitioned system can be developed according to certification standards. The second is how an existing single criticality system can be ported to this new mixed criticality architecture, with the aim of supporting both low criticality components, robust high criticality components and standard high criticality components; and how such a system can be utilised to combat the introduction of WCET pessimism.

## 4.1 Literature Survey

In the literature an MCS is a system which combines software of multiple DALs on the same processor. The technical objective of which is to provide sufficient evidence that a low criticality component cannot jeopardise any high criticality component's temporal or functional requirements, while still providing a level of service to the low criticality component. One approach to MCS development is to deploy the partitioned architecture defined by the ARINC 653 standard [22]. This standard defines a partitioned model principally aimed at the development of Integrated Modular Avionics (IMA), but capable of supporting partitions developed against different DALs. The issue with the ARINC 653 ap-

proach is that the solution defined for temporal partitioning, essentially a two-level scheduler with time division, makes the approach difficult to apply to a complex control system [23]. This is because it can lead to the introduction of higher completion jitter, longer end-to-end transaction response times and in general it can be difficult to accommodate a complex task schedule into fixed time partitions [23].

The advantage of an ARINC-653 system is that it provides an integrated whole system design, encompassing scheduling theory alongside wider system design, designed against certification guidelines. As explored further in this literature, this is to a certain extent missing from the existing literature on wider MCS papers.

This literature survey now considers three key areas of MCS design, firstly Scheduling Theory, where arguably the largest body of research has been conducted. Secondly, Schedulability Analysis, focusing in particular on two seminal papers, and finally wider System Design, where it is argued that further refinements to the existing literature is required.

### 4.1.1 Scheduling Theory

Vestal [24] was one of the first publications to consider the schedulability of a MCS. The work draws the comparison that the reliability of the WCET figure used for each task is commensurate to its criticality. This is based on the observation that lower DAL tasks are not developed, or verified, to the same rigour that higher DAL tasks are; and therefore the output WCET figures cannot be expected to be as reliable.

Vestal essentially introduces a model where a task can be considered to have multiple WCETs which individually provide a different degree of assurance that true execution times for a task will not exceed the analysed value. Where each assurance level is taken from the set $L \in$

$\{A, B, C, D, E\}$, it can be assumed that the WCET for task $\tau_i$ follows the rule $C_i^A \geq C_i^B \geq C_i^C \geq C_i^D \geq C_i^E$.

In practice each task will only be analysed to the level of assurance required for that task, with the aim being to only expend the amount of WCET analysis effort that is proportional to the required assurance. Assuming a dual criticality system - each high criticality task would be required to have two WCET figures - $C_i^{HI}$ and $C_i^{LO}$, whereas a low criticality task has one WCET figure[2] - $C_i^{LO}$.

Building off Vestal's work, Baruah et al. [25] introduced three models for Mixed Criticality Scheduling - partitioned criticality, Static Mixed Criticality (SMC) and Adaptive Mixed Criticality (AMC).

Partitioned criticality [25] (also referred to as Criticality Monotonic Priority Ordering) is the simplest form of mixed criticality scheduling, where priorities are assigned according to each task's criticality. Accordingly a task with a higher criticality will always be scheduled with a higher priority than another task of lower criticality. This approach should ensure a timing error in a low criticality task cannot affect the temporal requirements of a high criticality task, therefore requiring no run time monitoring. However, there is no consideration in the paper as to whether this approach is sufficient, and because the scheduler will always execute a high criticality task if one is ready, it makes it significantly more difficult to meet low criticality task deadlines. One advantage offered by the approach is that run-time monitoring may not be required, however as Baruah et al. [25] point out, many safety critical systems already incorporate this for the purpose of error detection.

SMC and AMC [25] on the other hand assign task priorities according

[2]Vestal's original model noted that a low criticality task would also have a $C_i^{HI}$, however in practise this may not be known.

to their temporal requirements, regardless of criticality. SMC allows low or high criticality tasks to execute up to their $C_i^{LO}$ or $C_i^{HI}$ respectively; but they are then prevented from executing further [26]. This offers a stop dead point where any task must cease executing and provides adequate protection for high criticality tasks from low criticality tasks.

The AMC protocol builds off this; however, whereas SMC de-schedules one task if it executes for longer than $C_i^L$, AMC de-schedules *all* low criticality tasks if *any* high criticality task executes for longer than its $C_i^{LO}$. While the original paper did not explicitly define a recovery point, an obvious route back to re-enabling low criticality tasks is to use the Idle Task or state of the system. This is referred to in this paper as AMC+ and is based around the simple mode change protocol in [69]. The AMC+ protocol is achieved through a scheduler mode change which is summarised in Figure 4.1.

The AMC protocol therefore offers the potential for exploiting assumed pessimism between a task's $C_i^{LO}$ and $C_i^{HI}$ because the system schedulability is assessed in the 'normal' mode using each task's $C_i^{LO}$. This supersedes SMC which uses high criticality tasks $C_i^{HI}$. However, the approach places an assumption that each task's $C_i^{LO}$ is credible, which in practise cannot be proven. Execution of a task beyond its $C_i^{LO}$ is assumed to be an extremely rare event. Were this not to be the case then the service offered to low criticality tasks would reduce significantly.

Bate et al. [70] explored AMC and offer a further improvement by taking account of slack and gain time provided by tasks that finish before their $C_i$ in order to delay the switch to high mode. Their Bailout protocol aims to provide better service for low criticality tasks by providing levels of degradation as well as a faster route back to normal service. The Bailout protocol uses the AMC response time analysis method, and

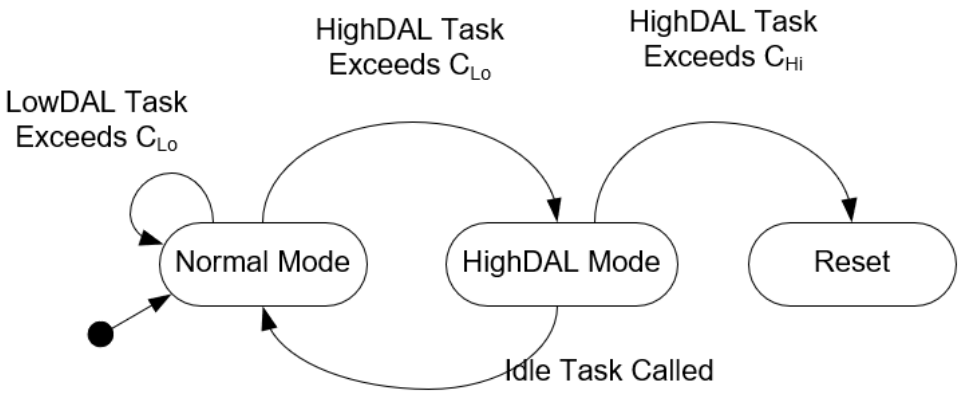


Figure 4.1: AMC+ State Flow Diagram.

therefore does not provide greater static schedulability over AMC. However, during dynamic analysis the Bailout protocol is shown to provide greater service for low criticality tasks.

The resilient model [34] utilises graceful degradation to improve low criticality task performance. The model employs resilience at the system level towards a certain number of timing faults, and robustness at the task level where certain 'robust' tasks are capable of skipping individual jobs when requested. Importantly, a task's robustness is independent from its criticality. Together these two techniques ensure the system does not resort to a state where low criticality tasks are denied service until absolutely necessary.

Burns et al. [34] introduced the following definitions for a robust mixed criticality system:

**Definition 4.1.** A *robust* task is one that can safely drop one non-started job in any extended time interval.

**Definition 4.2.** The *robustness* of a complete system is measured by its F count (how many job overruns can it tolerate without jobs being dropped or deadlines missed) and its M count (the number of job overruns

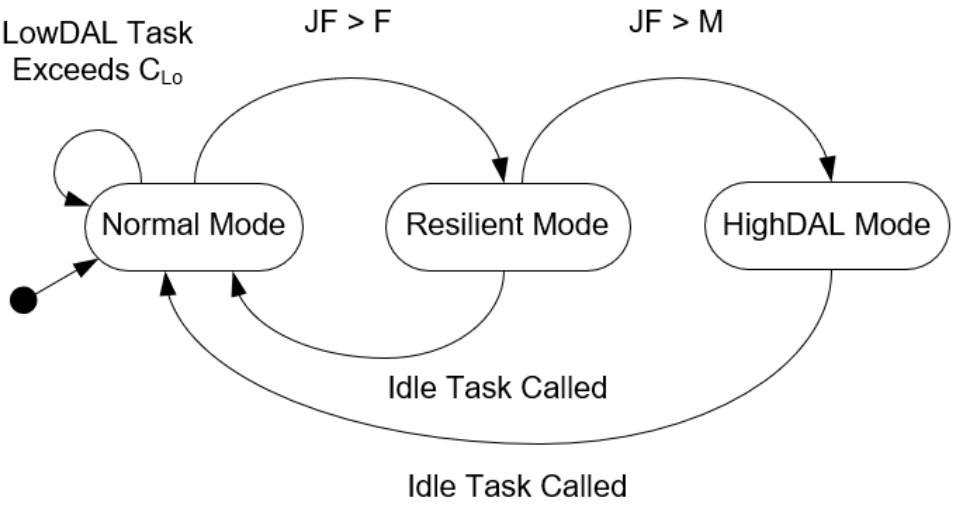


Figure 4.2: Resilient State Flow Diagram.

the system can tolerate once each robust task has dropped one job).

**Definition 4.3.** A *resilient system* is one that aims to achieve graceful degradation to adequately cope with more than M overruns.

**Definition 4.4.** A *fault* is measured when one task overruns its $C^{LO}$.

**Definition 4.5.** An *error* is the manifestation of one or many faults and represents the point where a task fails to adhere to its timing requirements.

In short, in this context a resilient system employs robustness to cope with one or many faults, while avoiding errors.

The resilient scheduling model introduced in [34] is capable of coping with $F$ faults, before reverting to the 'Resilient Mode' where robust tasks skip their jobs. At this point the system is capable of coping with further faults up to a total of $M$ faults, where $F < M$. Once the fault count increases above $M$, the system reverts to the 'High Criticality' mode,

as with the AMC model. Once the system reaches the idle state, the fault count is reset and, if required, the system reverts to the 'Normal Mode'. This is illustrated in Figure 4.2, where $JF$ provides a count of the number of job faults.

The schedulability analysis presented in the paper provides a proof that high criticality, robust tasks, comply with their schedulability requirements. The analysis also provides a bound on the number of jobs a robust task may skip between idle points. However, the analysis provides no guarantees on the service given to low criticality tasks, or indeed the time between individual robust task skip bursts. This is a problem that is explored in Chapter 5 of this thesis.

Finally, as opposed to the fixed priority techniques explored so far in this literature survey, a large body of work has also been produced focusing on dynamic priority algorithms, such as Earliest Deadline First based methodologies [71]–[73]. This thesis focuses on the development of a MCS to support a safety critical aerospace domain. While the debate about the most optimal approach rages, the principal requirement for a safety critical application is the development and proof of a safe system. Therefore, as fixed priority systems offer the simplest route to system verification and proof, they form the focus of this work.

### 4.1.2 Static Schedulability Analysis

The static schedulability analysis used for the AMC+ implementation was introduced by Baruah et al. [25]. The schedulability analysis is performed in three stages. The response time of each task is assessed in the high and then low modes. Finally, the response times of the high criticality tasks are assessed during a mode change from low to high.

As long as all tasks execute for less than their $C_{LO}$ then the system

remains in the low, or normal, mode. The low-criticality mode WCRT, $R_i^{LO}$, is calculated using Equation 4.1, where $hp(i)$ is the set of higher priority tasks than task $\tau_i$. The equation should be recursively solved until the input $R_i^{LO}$ matches the output $R_i^{LO}$.

$$R_i^{LO} = C_i^{LO} + \sum_{j \in hp(i)} \left( \left\lceil \frac{R_i^{LO}}{T_j} \right\rceil C_j^{LO} \right) \tag{4.1}$$

Should any task execute for longer than its $C_{LO}$ then the system will revert to the high mode, where only high criticality tasks are permitted to execute. The response time calculation for the high-criticality mode WCRT, $R_i^{HI}$, is shown in Equation 4.2; where $hpH(i)$ is the set of high criticality higher priority tasks than task $\tau_i$. Again this function should be recursively solved.

$$R_i^{HI} = C_i^{HI} + \sum_{j \in hpH(i)} \left( \left\lceil \frac{R_i^{HI}}{T_j} \right\rceil C_j^{HI} \right) \tag{4.2}$$

Low criticality tasks are not considered when in the high-criticality mode as they are de-scheduled by the system. The sufficient mode change analysis [25] then defines the response time analysis for a high criticality task during a low-to-high mode change as shown in Equation 4.3. In this equation $hpL(i)$ shows the set of low criticality higher priority tasks.

$$R_i^{*} = C_i^{HI} + \sum_{j \in hpH(i)} \left( \left\lceil \frac{R_i^{*}}{T_j} \right\rceil C_j^{HI} \right) + \sum_{k \in hpL(i)} \left( \left\lceil \frac{R_i^{LO}}{T_k} \right\rceil C_k^{LO} \right) \tag{4.3}$$

This ensures the interference from low criticality tasks is capped as $R_i^{*}$ must be greater than $R_i^{LO}$.

The focus of the MCS literature studied so far in this literature survey has been towards scheduling theory, this research has offered significant advances on the understanding on how such systems could be developed.

However, as highlighted by Ekberg and Yi [74], and by Altmeyer et al. [31] development of a MCS cannot be assumed to be restricted to development of the scheduler. MCS development must also take account of the wider scheduler and RTOS design and analysis, as well as examining what assumptions MCS operation places on the tasks within a system, particularly low criticality tasks. This is vital as development of a system employing components of differing criticalities cannot simply be viewed as a scheduling problem. The following section now explores existing papers that have focused on the development of MCS at the system level.

### 4.1.3 System Definition

This section now explores wider system issues that must be addressed to support MCS development, and identifies where the existing literature has tried to address these. Altmeyer et al. [31] examine what assumptions MCS development places upon WCET and highlight the issues which currently exist within WCET analysis techniques. It outlines how MCS developers currently lack the necessary information, or in some cases, confidence required to produce sound MCS systems. The paper essentially highlights how development of MCS scheduling techniques must not be considered in isolation, but instead must be considered as part of a wider system. The paper highlights one of the many holes that currently exist within the literature surrounding MCS development, in that MCS development places a greater emphasis on assessing WCET confidence.

One of the general issues with a number of the refinements made to the Vestal model is that the papers lacked details of how the implementation would ensure the properties needed for certification. The analyses for instance does not consider overheads as part of the analysis, and the papers do not consider how overheads could be reduced. Furthermore,

an analysis of what requirements each scheduling method places on the hardware and RTOS, in the way of required facilities, has not been explored. It is simply assumed these features are already present and that their overheads can be neglected. Finally, there has been very little work actually looking at the implementation of a MCS into a real system. Therefore, the existing literature leaves many unanswered questions about how effective a MCS can be in practice.

Looking at wider system development; Sousa et al. [75] identify the overheads induced by a multi-core task-split system. The work assesses each overhead source, and incorporates the overheads into the schedulability analysis for a slot-based system. However, this work does not progress far enough to allow full end to end system development. For instance, a method for measuring the identified overheads, or indeed minimising them, is not presented.

Freitag et al. [76] divides tasks of different criticalities across different cores on a multi-core processor, in order to simplify system proof. The system supervisor analyses the interference induced by low criticality cores on high criticality cores, disabling the low criticality core if required. Herman et al. [77] perform an analysis of the development of a mixed criticality multi-core system. However, the initial analysis does not progress far enough to support actual development, for instance through proof of the effect of overheads on the schedulability of the system. Finally, Paolillo et al. [78] examines the benefits of porting an industrial case study to a mixed criticality system; finding that the potential low criticality task utilisation is high, but also identifying how the identification of sound task WCETs had a significant effect on the service afforded to low criticality tasks. The paper however did not progress far enough to explore how such a system could be implemented and certified.

Several papers have looked at reducing the cost of system overheads in schedulable systems principally following two main approaches: firstly making the analysis less pessimistic and secondly reducing the actual overheads.

In terms of the analysis pessimism, a number of papers have focused on the area of Cache-Related Preemption Delays (CRPD) where an understanding is derived of the impact of on-the-cache contents and which parts of the software cannot preempt each other [79]. Alternatively, Davis et al. [80] consider how MCS schedulers can be modified to attempt to avoid extended context switching times thanks to the cost of changing processor mode. The paper is one of the first papers to consider temporal partitioning in the context of the additional overheads induced by spatial partitioning processor features. Additionally, Burns and Davis [81] explore how tasks in a MCS can invoke final pre-emption-blocking points in order to defer being preempted at a point when a task job is about to complete. The method is shown to produce a greater proportion of schedulable systems. However, what is unclear is how such a task set can be developed in practice, whether each task would need to be manually analysed and altered, or whether RTOS features are required to avoid preempting a task when it reaches a certain point in its execution.

Focusing on the reduction of RTOS overheads; overlooking the obvious aim that any RTOS or scheduler must be designed to be as efficient as possible; two main areas of research have been performed on reducing the occurrences of overheads. Firstly, a number of researchers have looked at minimising the number of priority levels, for example in the work by Audsley et al. [82], which has demonstrated that this process can lead to a reduction in the number of task context switches. The second approach is grouping a number of software components (tasks in the original

Rolls-Royce control software) into larger schedulable tasks (referred to here as SuperTasks). This approach is the same philosophy as adopted in AUTOSAR systems where runnables are grouped to form tasks as part of reducing overheads [83]–[85]. However, the approaches either ignore inter-task dependencies (transactions) [83], or require the possibility to duplicate tasks shared between transactions [85]. This means, for instance, the approach would not be suitable for the system introduced in Chapter 2.

### 4.1.4 Summary

A review of the available literature reveals that while there has been a large field of work completed on identifying appropriate, efficient scheduling methodologies; there has been less work focused on MCS design. This leaves a gap between what is possible and what is usable as the techniques presented fail to take account of real system effects such as inter-task relationships, or system overheads. The work that has been completed on MCS design, or on handling overheads, has either focused on reducing overheads through efficient system design, or at improving WCET processes. However, neither have necessarily tried to apply these techniques in an industrial project, or indeed, alongside the MCS scheduling methodologies discussed elsewhere.

In summary, the available literature has yet to study, and extend, MCS research towards a full end-to-end system integration, incorporating real system overheads, analysis techniques, and criticality requirements. The following section now aims to do this by examining how a system to allow mixed criticality integration can be designed, to fulfil certification requirements, to support a real industrial system. This includes how requirements for appropriate task partitioning can be fulfilled, and how

a system may be developed and analysed. The chapter then continues to explore how a legacy system can be automatically ported to this new MCS design.

## 4.2 Mixed Criticality System Design

This section discusses the development of a two-level MCS. Initially the certification requirements for a MCS, as guided by DO-178C [40], are introduced. The section then progresses to examine the features of a MCS designed to comply with these requirements.

In this thesis 'high criticality' refers to the highest criticality component permitted to execute upon the processor (nominally DAL A); whereas low criticality components refer to any components that have a lower criticality than the highest processor component (DALs B, C, D and E).

### 4.2.1 Certification Requirements

This section examines the certification requirements surrounding the development of a Mixed Criticality Scheduler; the target system studied being an aircraft engine control system. Accordingly, only the guidelines detailed in DO-178C [40] are explored in this section. However the guidelines are considered similar to those detailed in other software domains such as ISO26262 [86] and IEC61508 [21].

DO-178C Section 2.4 defines five requirements for partitioning as follows:

1. A partitioned software component should not be allowed to contaminate another partitioned software component's code, input/output (I/O), or data storage areas.

2. A partitioned software component should be allowed to consume shared processor resources only during its scheduled period of execution.

3. Failures of hardware unique to a partitioned software component should not cause adverse effects on other partitioned software components.

4. Any software providing partitioning should have the same or higher software level as the highest level assigned to any of the partitioned software components.

5. Any hardware providing partitioning should be assessed by the system safety assessment process to ensure that it does not adversely affect safety.

In essence DO-178C expects that in the absence of evidence to prove that a low criticality component may execute outside of its design defined boundaries, it is essential that protection is put in place to prove that promiscuous components cannot affect the wider system. This ensures that the evidence provided for high criticality components to show compliance against their requirements is still valid when co-located with low criticality components.

### 4.2.2 Partitioning

This section explores the design of a partitioned scheduler to support a Mixed Criticality System. One potential option would be to use the scheduling methodology defined by ARINC 653. However, this model is based on a strict time slicing model which is considered too restrictive for the implementation of a control system with extensive hardware in-

teraction, which relies on adherence to strict periodic execution against strict jitter requirements [23].

The assumption that should be made for a low criticality task is that the task may execute, if allowed, for longer than its observed HWM; its $C_i^{LO}$ is assumed to be optimistic. However, due to the rigorous testing regime the software undergoes and the extensive in-test and in-flight monitoring, it is known that it is rarely exceeded.

From a certification point of view as the evidence to prove otherwise may not have been produced to the same level as a high criticality component, then a certification authority must assume a low criticality component is more likely to contain an error. Thus, as guided by the requirements noted in Section 4.2.1, partitioning must be employed to prove that any errors that occur in a low criticality partition cannot propagate to a high criticality partition.

The scheduler proposed in this work implements two key protection mechanisms to implement a DO-178C partitioned architecture: the use of timer driven interrupts, and the use of processor memory protection. Figure 4.3 shows the statechart for the interrupt-driven scheduler. This is explored in more detail in the following subsections.

#### 4.2.2.1 Temporal Partitioning

A timer-driven interrupt is employed both to control the release of new tasks by invoking a scheduler tick, and to interrupt low criticality components when they reach their $C_i^{LO}$. As the interrupt handler prepares to switch in a task, one of the final operations is to set the interrupt timer to the lowest of either 1) the time to the next task release, or 2) in the case of a low criticality task, the allowed execution time remaining.

High criticality tasks are not regulated in the same way. If a high

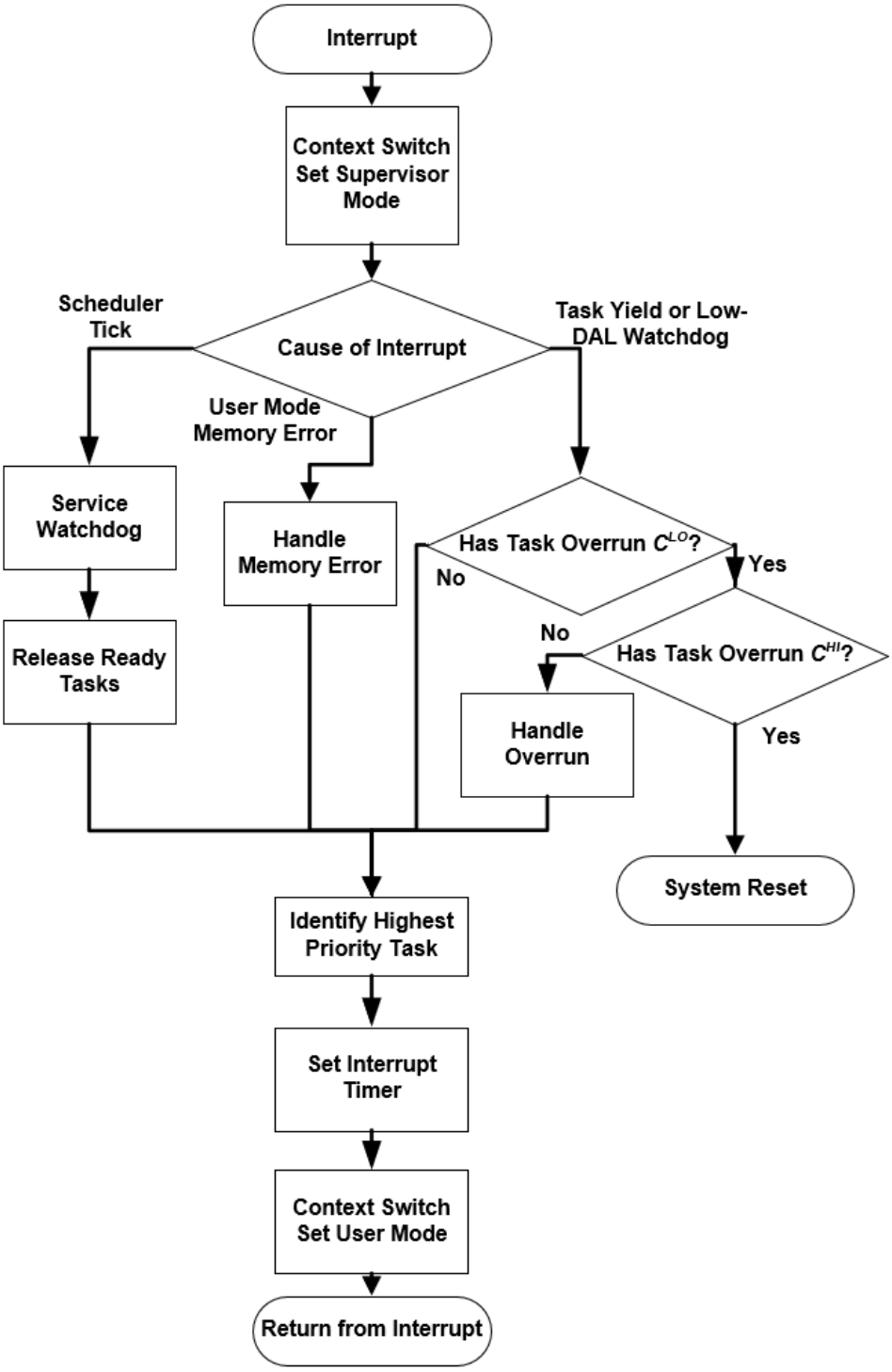


Figure 4.3: Partitioned Scheduler Statechart.

criticality task executes beyond its $C_i^{LO}$ then it is permitted to continue. However, the next time the scheduler executes it will identify the need to move into the high criticality mode. This is controlled by the 'Handle Overrun' block within Figure 4.3. The operation of the 'Handle Overrun' block is dependent on the scheduling methodology chosen. For instance, in the case of the AMC+ algorithm [69], the block will transition the system into the high criticality mode. When in high criticality mode, any low criticality tasks that are ready to execute will be suspended.

The 'Identify Highest Priority Task' block is then responsible for choosing which task should run next. The process simply chooses the highest priority task that is currently ready to execute.

Return to the normal mode is controlled by the idle task, which is developed to the highest criticality of the system. Crucially, low criticality tasks that have been suspended while the system executed in the high criticality mode are not released again until their next scheduled periodic execution. This is illustrated in Figure 4.4, which shows a simple example task set. At point A the highest priority high criticality task overruns its $C_i^{LO}$. Being a high criticality task it is permitted to continue, however when the task completes its job at point B the interrupt handler identifies the overrun and moves the system to the high criticality mode. This mode change blocks the release and execution of the low criticality task. Then at point C as the system enters the Idle task, the system mode is reverted back to the Normal mode. However, the low criticality task is not released again until point D, the next scheduled periodic execution point. This controlled execution resumption is important as it avoids inducing offsets which could affect the validity of the static schedulability analysis.

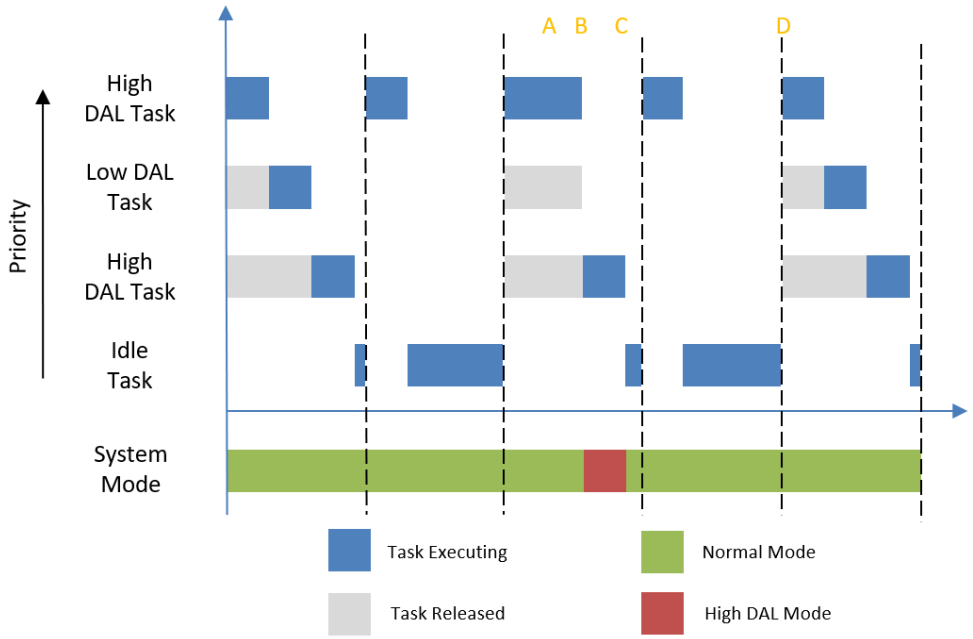


Figure 4.4: Example Partitioned Scheduler Operation.

#### 4.2.2.2 Spatial Partitioning

Processor memory protection is employed in a User/Supervisor arrangement. All tasks execute in a design-defined protected area of memory, with access to different hardware features or memory regions either permitted or restricted as necessary. Should any task execute outside these fixed boundaries, then an interrupt is raised and the interrupt handler handles the data error. This is enforced by the scheduler setting the proper User Mode when returning to a task, as illustrated in Figure 4.3.

The memory protection employed ensures that each component executing on the processor cannot execute outside its design time defined boundaries, thus providing protection for high criticality tasks from low criticality task promiscuous memory or hardware interactions.

#### 4.2.2.3 System Partition Assignment

Each software partition executes within its own area of memory, has a defined set of permitted hardware accesses, and has its own defined temporal requirements; which in the case of low criticality components, restrict the component's execution. In order to support this type of execution the following design constraints must be placed upon each partition:

- Each partition represents a thread-safe self contained execution unit.
- Each partition contains software developed to the same criticality level.
- No partition can *arbitrarily* enter a critical section, or block an interrupt from occurring.

Arguably the final constraint is a significant restriction. Partitions may need to use critical sections or block interrupts from occurring, for example for performing writes to hardware, but these must be performed through top level system calls, where the application of such operations can be controlled and analysed.

Where a partition needs to send a message or communicate with another task, this communication is handled through a defined memory interface which is both developed to the highest DAL and contains protections to constrain any task blocking or priority inheritance. The worst case blocking time must then be analysed and incorporated into the system schedulability analysis.

The approach for critical operations used in this application relies on a simple critical section entry and exit approach, where the RTOS provides mechanisms to temporarily disable interrupts. This allows simple

analysis to be performed to provide a bounded blocking time, which can be confirmed will not inhibit key system utilities such as watchdog servicing. This approach is pessimistic from a timing point of view, however in the system targeted, where inter-partition and inter-task dependencies are already minimised, the approach is sufficient. Were a more complex system with significant inter-dependencies need to be ported to this architecture, then a more complex, less pessimistic, approach may be required.

### 4.2.3 Derivation of Task Timing Parameters

The derivation of appropriate $C_i^{LO}$ and $C_i^{HI}$ figures for each task is vital to ensure the schedulability of the system can be properly assessed [31]. This section explores both the assumptions placed on $C_i^{LO}$ and $C_i^{HI}$, as well as how each parameter can be defined.

A number of mixed criticality scheduling methodologies, such as those defined from the AMC model [25], have the same underlying assumption - that all tasks will complete by the $C_i^{LO}$, and that any task exceeding $C_i^{LO}$ can be treated as a one off fault. The AMC protocol itself for instance will stop all low criticality tasks if any high criticality exceeds its $C_i^{LO}$. Therefore an optimistic $C_i^{LO}$ will lead to little or no service for low criticality tasks.

For high criticality tasks the analysed $C_i^{HI}$ represents an absolutely sound WCET figure. The schedulability analysis, and partitioning scheme, assume this time is never exceeded.

Chapter 3 established a process for the early automated identification of system timing parameters producing two timing parameters for each analysed task. In the context of a mixed criticality system - the raw HWM times derived directly from execution of the search algorithm

would be used for a task's $C_i^{LO}$. Whereas the analysed results once input into the hybrid measurement based WCET analysis tooling would be used for the task's $C_i^{HI}$.

Furthermore, the development of safety critical aircraft engine control systems for certification requires an extensive test regime to be followed. This includes considerable integration testing, which provides as an output system level HWM times and measurements for WCET analysis. These HWMs are used at Rolls-Royce to build further confidence that, throughout an entire test campaign covering thousands of flight cycles, no larger task execution time has been observed. This HWM is therefore well suited to supplement a task's $C_i^{LO}$, with the analysed WCET able to supplement each task's $C_i^{HI}$ as an engine development programme matures. The $C_i^{LO}$ figure can be said to be sound, but has not been proven not to be optimistic. While the $C_i^{HI}$ is assumed to be pessimistic, but sound. The advantage of this approach is that the figures used for each task's $C_i^{LO}$ and $C_i^{HI}$ improve throughout the software design life-cycle; from initial results provided through the work in Chapter 3, up to a certifiable result as a project approaches certification deadline.

### 4.2.4 RTOS and Target Hardware Requirements

In order to provide a system that enables partitioned operation and is capable of being certified, the RTOS and target hardware must provide a defined set of features. From the point of view of the target hardware, the processor must provide hardware support for invoking and controlling timer driven interrupts, as well as a controllable memory protection unit (MPU) to provide the required spatial partitioning.

This work is designed to be RTOS agnostic, with the RTOS simply invoking the scheduler, and providing the required utilities, such as

mechanisms to suspend the release of tasks, or force the completion of already executing jobs - both required to control the execution of low criticality tasks. However, it is important that the RTOS overheads are not only easily calculated, but that they are also linearly proportional to the number of tasks in the system. This ensures the overheads of the RTOS and scheduler can be easily incorporated into the schedulability analysis of the system at design time.

Finally, the target hardware, RTOS and all protections that they provide, must be developed according to the highest DAL of the system.

### 4.2.5 Schedulability Analysis Extensions

One of the shortfalls identified in the existing literature surrounding MCS schedulability analysis was the inclusion of the RTOS and scheduler overheads. This section now explores how the overheads of such a system can be broken down, and how the AMC schedulability analysis equations need to be expanded to support inclusion of the identified overheads.

In order to include the execution time of the scheduler shown in Figure 4.5 into the response time analysis for the system, the overheads were broken down into three constituent parts as described below:

1. Tick Overhead - $\delta_T$ (Figure 4.5 - dot/dash line, red). It includes:
    - The pre-emption of the executing task.
    - The handling of system services, e.g. the watchdog.
    - The context switch and calling of the highest priority task.
    - The release of any tasks into the ready state. Measured separately as $\delta_R$. (Figure 4.5 - solid line, green).
2. Start Task Time - $\delta_S$ (Figure 4.5 - dashed line, blue). It includes:

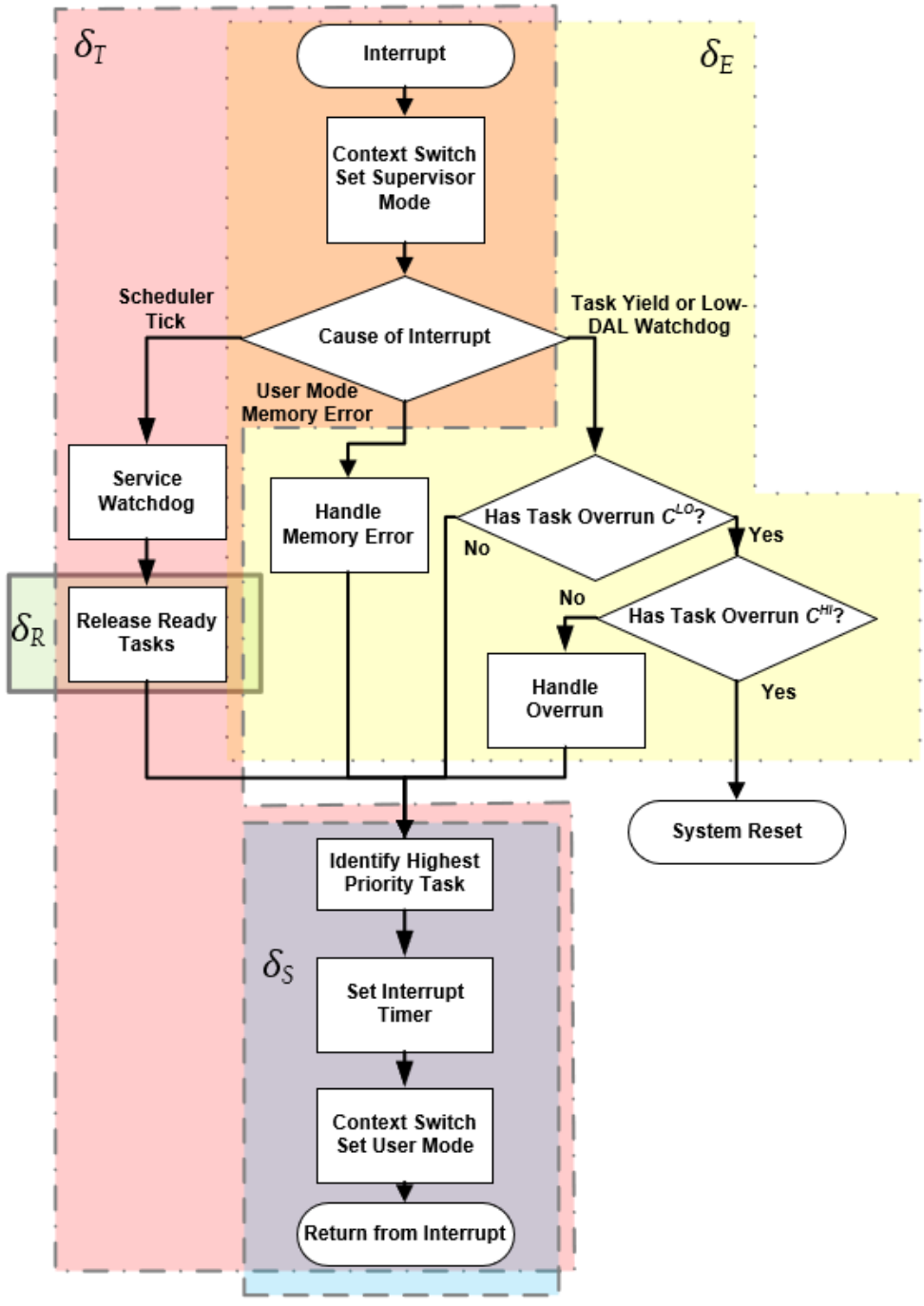


Figure 4.5: Partitioned Scheduler Statechart with Overheads.

- The initial time taken to context switch each task into the executing state. Except for the highest priority task, which is accounted for in the tick overhead.

3. Stop Task Time - $\delta_E$ (Figure 4.5 - dotted line, yellow).

    - The end time taken when a task finishes executing and returns to the scheduler.

Task releases were fixed to only occur on a scheduler tick, and the scheduler tick is the only component that can interrupt another task. The execution time of each overhead was measured during normal system operation, which included at certain points, the schedule's critical instance. This ensured the maximum execution time for each overhead was captured by ensuring observation of the instance where the maximum number of tasks are moved into the released state.

The release overhead was measured and recorded against the number of tasks being released. This allowed the release overhead of each task to be assessed, which supported the design principle that the system overheads should either be linear, or less than linear, against the number of tasks being released.

The credibility of this maximum observed overhead is based on the following implementation details:

- The use of a time deterministic target processor. As introduced in Chapter 2.

- Tasks are only released on the system tick. The system tick period is equal to the greatest common divisor of the tasks' period. All other task periods in the system are a harmonic of the tick period.

- Each overhead is measured while the system executes a full system test campaign on a full simulation rig.
- The RTOS is carefully designed to ensure the task release overhead is linearly proportional to the number of tasks in the system.

Finally, each overhead was factored into the analysis through synthetic tasks, in the same way originally introduced by Burns et al. [87], which was discussed further in Section 4.1.2. This method of essentially viewing certain overheads as tasks provides a safe and suitable method for taking account of the periodicity of the overheads. It allows the overheads to be placed at the appropriate place in the schedule to ensure correct analysis of interference.

The target for this assessment is on the AMC algorithm, originally presented by Baruah et al. [25]. In order to aid the following discussion the equations for the response time analysis of each task in the low criticality mode, high criticality mode and during a mode change are shown in Equations 4.1, 4.2 and 4.3.

$$R_i^{LO} = C_i^{LO} + \sum_{j \in hp(i)} \left( \left\lceil \frac{R_i^{LO}}{T_j} \right\rceil C_j^{LO} \right) \tag{4.1}$$

$$R_i^{HI} = C_i^{HI} + \sum_{j \in hpH(i)} \left( \left\lceil \frac{R_i^{HI}}{T_j} \right\rceil C_j^{HI} \right) \tag{4.2}$$

$$R_i^{*} = C_i^{HI} + \sum_{j \in hpH(i)} \left( \left\lceil \frac{R_i^{*}}{T_j} \right\rceil C_j^{HI} \right) + \sum_{k \in hpL(i)} \left( \left\lceil \frac{R_i^{LO}}{T_k} \right\rceil C_k^{LO} \right) \tag{4.3}$$

The effect that the tick overhead has on the response time of a task can then be calculated as follows:

$$\delta_T^{MODE} = \left\lceil \frac{R_i^{MODE}}{T_{TICK}} \right\rceil C_{TICK} + \sum_{j \in MODE(i)} \left( \left\lceil \frac{R_i^{MODE}}{T_j} \right\rceil C_{REL} \right) \tag{4.4}$$

In Equation 4.4, as in the following Equations 4.5 and 4.6 the value of $R_i^{MODE}$ used should either be $R_i^{LO}$, $R_i^{HI}$ or $R_i^*$ depending on whether the low mode, high mode or mode change response time is being calculated.

Secondly, the set of higher priority tasks used in each equation (denoted as $j \in MODE(i)$ or $j \in hpMODE(i)$) should be limited to those tasks permitted to execute in order to avoid undue pessimism. The scheduler tick occurs in all scheduler modes, as well as during a mode change. However the release overhead for low criticality tasks will only occur in the low mode, or during a mode change from low criticality to high. This low criticality task release overhead cannot be ignored during a mode change because the release of these tasks occurs before the highest criticality task begins to execute.

The start and stop overheads of each task are calculated as follows:

$$\delta_S^{MODE} = \sum_{j \in hpMODE(i)} \left( \left\lceil \frac{R_i^{MODE}}{T_j} \right\rceil C_{START} \right) \tag{4.5}$$

$$\delta_E^{MODE} = \sum_{j \in hpMODE(i)} \left( \left\lceil \frac{R_i^{MODE}}{T_j} \right\rceil C_{END} \right) \tag{4.6}$$

Equations 4.1 and 4.2 can therefore be extended as follows:

$$R_i^{LO} = C_i^{LO} + C_{START} + \delta_T^{LO} + \sum_{j \in hp(i)} \left( \left\lceil \frac{R_i^{LO}}{T_j} \right\rceil C_j^{LO} \right) + \delta_S^{LO} + \delta_E^{LO} \tag{4.7}$$

$$R_i^{HI} = C_i^{HI} + C_{START} + \delta_T^{HI} + \sum_{j \in hpH(i)} \left( \left\lceil \frac{R_i^{HI}}{T_j} \right\rceil C_j^{HI} \right) + \delta_S^{HI} + \delta_E^{HI} \tag{4.8}$$

Finally Equation 4.3 can be extended as follows:

$$R_i^* = C_i^{HI} + C_{START} + \delta_T^* + \sum_{j \in hpH(i)} \left( \left\lceil \frac{R_i^*}{T_j} \right\rceil C_j^{HI} \right) + \sum_{k \in hpL(i)} \left( \left\lceil \frac{R_i^{LO}}{T_k} \right\rceil C_k^{LO} \right) + \delta_S^* + \delta_E^* \quad (4.9)$$

This overhead model is built on two key assumptions. Firstly, that the overheads of the system are understood; and secondly, that a switch from a one criticality task to a different criticality task, and the associated context switch, takes the same time as switching between tasks of the same criticality. The first requirement is key for any safety critical system and is therefore deemed an acceptable restriction. If the second assumption cannot be fulfilled, or if the system must support a more complex context or thread switching mechanism then the overhead model, and the associated priority assignment mechanism may need to be adapted in a similar way to the work presented by Davis et al. [80].

Finally, based on the partition assignment requirements introduced in Section 4.2.2.3, no task is able to arbitrarily block an interrupt from occurring. Therefore, no blocking term is included in these equations. Should this be required, then the equations could simply be extended by adding the worst case system blocking time ($B$) to the start of each equation.

### 4.2.6 Review Against Certification Requirements

The system design notes highlighted so far are now discussed against the certification requirements identified from DO-178C. The system design is discussed in the context of executing within the Current System introduced in Chapter 2.

1. A partitioned software component should not be allowed to contam-

inate another partitioned software component's code, input/output (I/O), or data storage areas.

- *The control system software is developed and tested against appropriate standards which for business-critical as well as safety-critical reasons can be expected to comply with its requirements.*
- *The memory and hardware areas that are permitted for each partition are defined at design time, and passed to the RTOS at initialisation.*
- *The RTOS controls the target processor's user/supervisor mode configuration in order to regulate the operation of each partition; ensuring any memory transgressions are stopped and handled appropriately.*

2. A partitioned software component should be allowed to consume shared processor resources only during its scheduled period of execution.

   - *All low criticality tasks are bounded by the target hardware's scheduler controlled timing interrupt, which when calling a low criticality task will be set to interrupt the task when it reaches its $C_{LO}$.*
   - *High criticality tasks are not interrupted except by a scheduler tick. This is based on the trusted WCET analysis process followed for high criticality tasks. That is, a high criticality task's $C_{HI}$ is trusted.*

3. Failures of hardware unique to a partitioned software component should not cause adverse effects on other partitioned software com-

ponents.

- *If a failure prevents the software timing interrupt providing the expected protection then the hardware timing watchdog, which is accepted and proven in use, combined with an independent two-lane (duplex) architecture, will ensure acceptable safety.*
- *The spatial partitioning employed shall ensure a task cannot interact with address regions outside of its permitted bounds*
- *A high criticality component cannot rely on data from a lower DAL component for its safe operation.*
- *Hardware components are certified to at least the same standard as that of their software driver equivalents.*

4. Any software providing partitioning should have the same or higher software level as the highest level assigned to any of the partitioned software components.

   - *The RTOS, interrupt handler, scheduler and software timing watchdog should all be developed to the highest DAL and are executed as protected 'supervisor' mode components.*

5. Any hardware providing partitioning should be assessed by the system safety assessment process to ensure that it does not adversely affect safety.

   - *The processor, including the timing and memory supervision components, have been verified to DO-254 DAL-A and have been used on multiple certified systems.*

This section aimed to discuss the design of a partitioned scheduler to support a mixed criticality system. The scheduler aims to comply with

the certification guidelines presented in DO-178C. The next section now progresses to discuss how an existing system can be ported to the MCS discussed in this section.

## 4.3 Current Rolls-Royce Approach to Scheduling

This section builds off the Current System definition already provided in Chapter 2; and provides system information and background relevant to the partitioned system design discussed throughout this chapter.

'Visual Fixed Priority Scheduler' (VisualFPS) is a task attribute assignment and scheduling analysis tool framework developed initially by Bate and Burns [39] and then used by Rolls-Royce on all their FADECs since 2002 [15].

The current FADEC approach features a non-preemptive scheduler where all tasks are released by a clock tick which has a period equal to the greatest common divisor of the tasks' periods [39]. Timing protection is provided by a hardware timing watchdog that counts down from the clock tick period. If the counter is not reset before it reaches zero then the processor is reset, re-initialising the system. When combined with a dual lane architecture each with independent power supplies, sensors and actuators, the use of a hardware timing watchdog ensures the likelihood of a processor or software fault leading to a hazardous safety event is acceptably low.

From an industrial perspective, an aim of this work is to change the processing platform, scheduling mechanisms and tooling by only the minimum amount necessary; as the tooling is well understood and accepted by engineers and certification authorities respectively.

The example used for this analysis has already been certified as a DAL-A system. The system consists of a large number of tasks, each of which has a measured HWM and an analysed WCET, obtained using a hybrid-measurement based approach [13][37][67]. The HWM and WCET were used for the $C_i^{LO}$ and $C_i^{HI}$ respectively.

The timing requirements for the task set include independent task requirements of period $T_i$ (taken from the set [2.5, 5, 10, 12.5, 25, 50, 100, 200, 500]ms), deadline $D_i$ and in approximately 5% of cases; completion jitter $J_i$. Approximately 50% of tasks form part of a transaction, which may consist of between two and eleven tasks. The transaction requirements are further complicated by two factors. Firstly, some tasks appear in more than one transaction; and secondly, within a transaction it may be the case that some tasks have different periods. For example, a transaction may run a sequence of tasks with periods of 25, 50, 50, 25, 100 and then 25 respectively. An important decision taken is to use a repeatable algorithm (i.e. one that always produces the same results) that takes all the requirements and uses them to calculate the deadline for each task. Task priorities are then assigned using the Deadline Monotonic Priority Ordering (DMPO) algorithm where the task with the shortest deadline is given the highest priority. If all deadlines are met, all the timing requirements are met. The method ensures the schedule is correct by construction. This approach has a further advantage, key to industry, that by incorporating the timing requirements for each task into its design-time calculated deadline; the system can be easily proved, reviewed and understood by engineers and system integrators [39].

Finally, with respect to inter-task data transactions; each task is designed to communicate with a common interface, and follows a format of input-process-output. Furthermore, tasks are designed to execute upon

the data that is currently available and will not wait until fresh data is available. Where fresh data is required to move between tasks this is generally controlled by a transaction. This approach has the advantage of simplifying access to shared resources between tasks, and is therefore not considered further as part of this work.

In order to produce a set of mixed criticality tasks to integrate into the new MCS, a number of low criticality tasks were added to the task set. These tasks were chosen to mimic lower criticality monitoring functionality, which at present is distributed across different processing nodes in the control system. These additional monitoring functions took the total number of tasks in the system to 228.

### 4.3.1 Open Source Industrial Example

The control system design discussed so far throughout this section is taken from a commercially sensitive aircraft engine control system, and therefore cannot be discussed in detail in an open document. Hence, while the larger system introduced so far represents the end target system, this subsection introduces an open source industrial example which provides some of the same features as the commercially sensitive control system, but which can be discussed in more detail.

The initial work on the VisualFPS scheduling scheme, as discussed in Bate's thesis [38], provided an example control system task set. The task set is introduced in Table A.1 in Appendix A. The intertask transaction set is shown in Figure 4.6. The task set is used as originally presented with two modifications. Firstly, in order to provide a task set with multiple independent transactions sets, the three transactions highlighted in red were deleted. Secondly, four low criticality tasks were added to the schedule as denoted at the end of Table A.1 (Tasks P72_lo, P73_lo, P74_lo

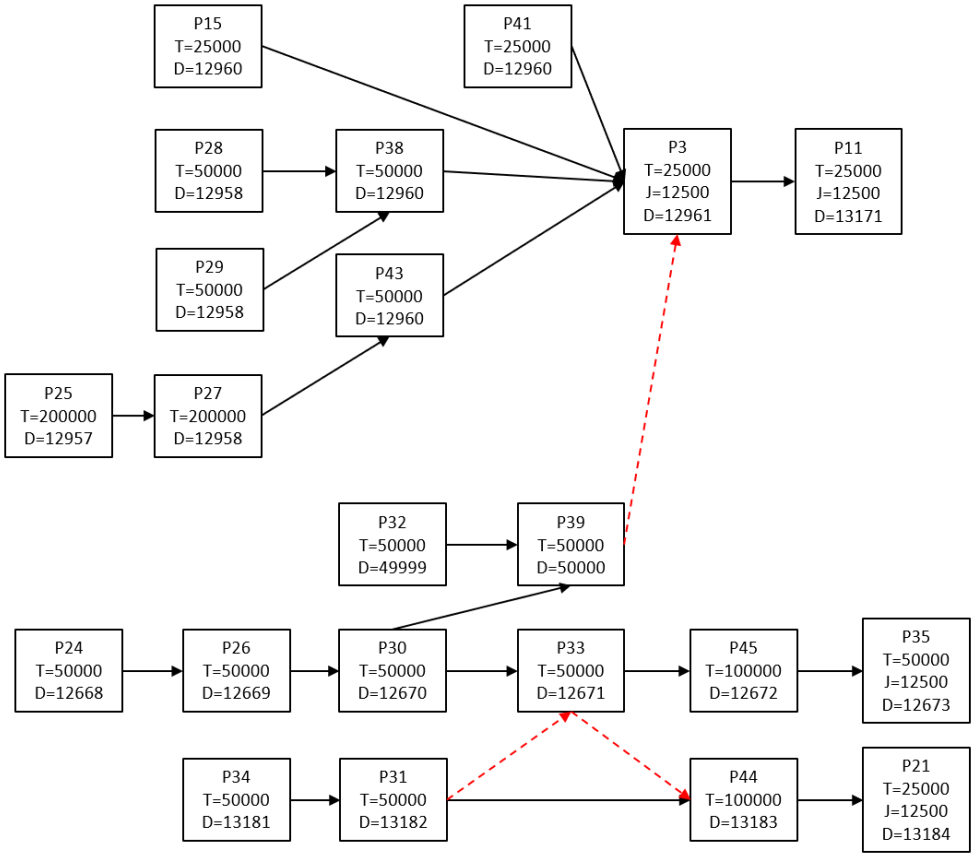


Figure 4.6: Example Control System Transaction Set [38].

and P75_lo). The addition of these four tasks (overlooking their different criticality) makes the task set un-schedulable when analysed using the existing VisualFPS process.

The process introduced by Bate [38] (which has since been further developed and used by Rolls-Royce) statically calculates the deadline for each task taking into account Period, Jitter and Transaction requirements. By then prioritising using the DMPO technique this offers a system which can be easily proven by design. Table A.1 shows the calculated deadlines for each task, with each deadline being calculated using Algorithm 3.

**ALGORITHM 3:** Calculating Each Task's Deadline.

```
1:  /* Initially set each task's deadline to equal its period */
2:  for i ∈ τ_i do
3:      D_i = T_i
4:  end for
5:
6:  /* If a task has a jitter requirement, then set the deadline to the lower of the
    deadline, or the jitter plus WCET */
7:  for i ∈ τ_i do
8:      if D_i > J_i + C_i then
9:          D_i = J_i + C_i
10:     end if
11: end for
12:
13: /* Iterate over each transaction, in reverse order, to ensure that each task
    preceding another in a transaction has a lower deadline */
14: while TaskSetChanging do
15:     TaskSetChanging = FALSE
16:     for j ∈ TransactionSet do
17:         for i ∈ reverse(Transaction_j) do
18:             if i == Transaction'_j LAST then
19:                 k = i
20:             else if D_i ≥ D_k then
21:                 D_i = D_k − 1
22:                 TaskSetChanging = TRUE
23:                 k = i
24:             end if
25:         end for
26:     end for
27: end while
```

The algorithm first sets each task's Deadline to its Period. It then iterates over each task that has a Jitter requirement and calculates a new Deadline for the task based on the task's Jitter requirement and WCET. Finally, the algorithm iterates over each Transaction and ensures that each task in a Transaction has a lower Deadline than each task that succeeds it in the same Transaction.

## 4.4 Porting Existing System to the MCS Architecture

The development costs involved in producing safety critical software are so great that any update, whether to tooling infrastructure or to the architecture of the system itself, must support legacy software as a pre-requisite. It is therefore important to understand how a legacy, non pre-emptive system such as the current Rolls-Royce control system can be ported to the new scheduler. Porting such a system also allows a study of the benefits of a real MCS industrial example to be analysed.

From an architectural requirements point of view there are a number of key differences between a non pre-emptive and a pre-emptive system. In particular, from a timing perspective, a non pre-emptive system is susceptible to blocking from large tasks; whereas a pre-emptive system, if not designed carefully, can suffer from larger overheads.

The current Rolls-Royce control system architecture consists of a large number of tasks, carefully designed to reduce the effects of task blocking in the current non pre-emptive scheduler (Section 4.3). The overhead assessment and implementation rules defined in Section 4.2.5 illustrated how the approach of using a large number of individual scheduled tasks is less desirable for a pre-emptive model. This is because firstly the

RTOS overheads of the pre-emptive RTOS are significantly higher than the overheads of the non-pre-emptive system due to the introduction of context switching and MCS task monitoring; but also because the overheads increase with the number of tasks, and their associated releases in the system. By reducing the number of tasks called from the RTOS, the number of tasks $MODE(i)$ is reduced to $MODE_{SUPER\_TASK}(i)$ (referring to Equation 4.4 where $MODE_{SUPER\_TASK}(i) << MODE(i)$), reducing the overhead induced by task releases. Furthermore, the number of higher priority tasks $hpMODE(i)$ is reduced in Equations 4.5 and 4.6, reducing the start and stop task induced overheads.

The aim of this process is therefore to take a set of control system tasks and to efficiently and appropriately port them over to a set of RTOS SuperTasks.

**Definition 4.6.** A SuperTask is an RTOS called function, constructed using one or many tasks.

There are two aspects that must be considered when porting components from one architecture to another. The first is the correct handling and protection of data transfers that are conducted across the system. The second is in the correct allocation of tasks to fulfil the temporal requirements of the compiled system. This thesis is concerned principally with the latter. The former, which primarily revolves around utilising a strict input-process-output architecture, is considered in parallel work.

### 4.4.1 Porting Tasks Without Clustering

In order to provide an initial state for comparison and potential improvement, the existing Rolls-Royce and open-source systems were ported directly over to the new pre-emptive MCS. Firstly, each task was directly

ported over to create an RTOS SuperTask. This set of RTOS SuperTasks were then prioritised using the DMPO scheme. Finally, the overheads for the system were calculated using the methodology set out in Section 4.2.5.

The maximum computation time for the RTOS and MCS run time monitoring system was analysed as the following:

- $C_{TICK} = 35\mu s$
- $C_{REL} = 7\mu s$
- $C_{START} = 25\mu s$
- $C_{STOP} = 30\mu s$

These figures were measured using the Rolls-Royce qualified process as defined by [13]. These overheads were used for the Rolls-Royce and Open source example systems discussed in the following sections. However, in Section 4.4.8 it is shown that the trends in the results and the contributions of the chapter are still applicable for varying overheads.

#### 4.4.1.1 Open Source System

The results of applying the schedulability analysis introduced in Section 4.2.5 to the open source system are shown in Tables 4.1 and 4.2. The results show how the system suffers from over 10% overheads which leads to a significant number (30%) of tasks failing their schedulability analysis.

#### 4.4.1.2 Rolls-Royce System

The results from porting the Rolls-Royce control system to the new MCS are shown in Tables 4.3 and 4.4. In this case the RTOS overheads measured 45% of total system utilisation, creating a system where in excess of 75% of tasks failed their schedulability analysis.

| | #SuperTasks | Schedulable Tasks | Transaction Pass? |
|---|---|---|---|
| NoClustering | 75 | 70.7% | Yes |

Table 4.1: Clustering Results When Applied to an Open Source Engine Control Case Study.

| | $\delta_S$ | $\delta_E$ | $\delta_T$ | $\delta_{SUM}$ |
|---|---|---|---|---|
| NoClustering | 3.8% | 4.6% | 2.5% | 10.9% |

Table 4.2: Clustering Results When Applied to an Open Source Engine Control Case Study.

It is clear that porting the existing control systems to a MCS pre-emptive system produces a system no longer able to comply with its requirements. Therefore, the following sections now examine how this system can be ported in such a way that allows it to still maintain its temporal correctness.

### 4.4.2 Clustering to Support System Design

The principal aim of clustering the set of control system tasks is to create a system that complies with its temporal requirements following execution of response time analysis. Based on the temporal requirement set, and on the design objectives of the MCS discussed in Section 4.2, the following success factors can be identified to confirm a system is schedu-

| | #SuperTasks | Schedulable Tasks | Transaction Pass? |
|---|---|---|---|
| NoClustering | 228 | 24.1% | Yes |

Table 4.3: Clustering Results When Applied to the Rolls-Royce Aircraft Engine Control System.

| | $\delta_S$ | $\delta_E$ | $\delta_T$ | $\delta_{SUM}$ |
|---|---|---|---|---|
| NoClustering | 17.6% | 21.1% | 6.3% | 45.0% |

Table 4.4: Clustering Results When Applied to the Rolls-Royce Aircraft Engine Control System.

lable:

- No SuperTask can contain two or more tasks of different criticalities. Thus ensuring the MCS rules surrounding partitioning can be maintained at the RTOS level.

- The response time of each individual task must be less than the task's calculated deadline. The deadline is calculated using Algorithm 3.

- For each transaction, any task preceding another task should have a higher priority than the succeeding task.

The basic algorithm for clustering a set of tasks is shown in Algorithm 4. The first part of the algorithm to Line 4 creates an ordered set of tasks from highest priority to lowest priority. The method for creating this ordered set on Line 4 is explored and analysed in the following subsections.

Once the ordered set of tasks have been identified the algorithm steps through the task set in priority order and divides tasks into SuperTasks, this process is shown in Algorithm 5. The first task is automatically placed in the first SuperTask. This SuperTask takes on the period, deadline and criticality of the task. All successive tasks are placed in the same SuperTask if the following rules apply:

**ALGORITHM 4:** Task Clustering Algorithm.

```
1: /* Calculate task deadlines using Algorithm 3 */
2: UnOrderedTasks = CalculateTaskDeadlines(Period, Jitter, Transactions)
3: /* Create OrderedTaskSet according to the defined clustering method */
4: OrderedTaskSet = OrderTaskSet(ClusteringMethod)
5:
6: Move OrderedTaskSet[0] into SuperTask[0]
7: SuperTask[0].Period = OrderedTaskSet[0].Period
8: SuperTask[0].Criticality = OrderedTaskSet[0].Criticality
9: SuperTask[0].WCET = OrderedTaskSet[0].WCET
10:
11: j = 0
12: for i in 1..OrderedTaskSet.Length do
13:     if AddToSuperTask(j, i) then
14:         Move OrderedTaskSet[i] into SuperTask[j]
15:         SuperTask[j].Deadline =
            Min(OrderedTaskSet[i].Deadline, SuperTask[j].Deadline)
16:         SuperTask[j].Period =
            GreatestCommonDivisor(OrderedTaskSet[i].Period, SuperTask[j].Period)
17:         SuperTask[j].WCET = OrderedTaskSet[i].WCET + SuperTask[j].WCET
18:     else
19:         j++
20:         Move OrderedTaskSet[i] into SuperTask[j]
21:         SuperTask[j].Period = OrderedTaskSet[i].Period
22:         SuperTask[j].Deadline = OrderedTaskSet[i].Deadline
23:         SuperTask[j].Criticality = OrderedTaskSet[i].Criticality
24:         SuperTask[j].WCET = OrderedTaskSet[i].WCET
25:     end if
26: end for
27:
28: Apply DMPO to SuperTask set
```

**ALGORITHM 5:** AddToSuperTask(SuperTask j, Task i).

```
1: /* Identify Whether Task(i) can Join SuperTask(j) */
2: if OrderedTaskSet[i].Period is harmonic of OrderedTaskSet[i-1].Period
   and OrderedTaskSet[i].Criticality == SuperTask[j].Criticality
   and SuperTask[j].Period ≥ (OrderedTaskSet[i].WCET + SuperTask[j].WCET)
   then
3:    return true
4: end if
5: return false
```

- The task's period is a harmonic of the SuperTask period, or the SuperTask's period is a harmonic of the task's period.
- The task's criticality is the same as the SuperTask's criticality.
- The addition of a task into the SuperTask will not increase the SuperTasks total execution time to more than its period.

If any of these rules fail, then the task is placed into a new SuperTask. The SuperTask then assumes the lowest period and deadline of the tasks inside the SuperTask. Any task with a greater period than its SuperTask period is placed inside a static counter controlled conditional statement which ensures the task is only executed on its period. For instance a 50ms task inside a 25ms SuperTask will execute every other invocation of the SuperTask.

Once this allocation of tasks to SuperTasks has completed, the set of SuperTasks is prioritised using the DMPO scheme.

The following sections identify and investigate different methods for ordering the task set (**OrderTaskSet(ClusteringMethod)** from Algorithm 4). These methods focus on the temporal requirements of the task set. Parameters such as the task's criticality were not used for ordering the task set in order to comply with the requirements of the MCS design.

### 4.4.3 Porting Tasks By Period

The first method of task prioritising uses the Rate Monotonic Priority Ordering scheme [88] by setting the Deadline of each task to its Period. The algorithm for performing this operation is shown in Algorithm 6. As indicated, all tasks are ordered and that ordered set is broken down into SuperTasks based purely on their period.

**ALGORITHM 6:** OrderTaskSet(Period).

```
1:  /* Order tasks by Period */
2:  OrderTaskSet = UnOrderTaskSet
3:  while TaskSetChanging do
4:     TaskSetChanging = FALSE
5:     for i ∈ OrderTaskSet do
6:        for j ∈ (i...τ_j) do
7:           if T_j < T_i then
8:              OrderTaskSet = SwapTasks(i, j)
9:              TaskSetChanging = TRUE
10:          end if
11:       end for
12:    end for
13: end while
14: return OrderTaskSet
```

Figure 4.7 shows the result of the task clustering by period algorithm on the open source industrial example. The figure focuses on the transactions present in the system, all tasks not listed (and not part of a transaction) are ordered according to their period in the same way.

The figure shows how the algorithm risks breaking transactional requirements, as for instance SuperTask 1 executes before SuperTask 2 and 4; despite SuperTasks 2 and 4 containing preceding transactional tasks to SuperTask 1's in the transaction shown at the top of the figure.

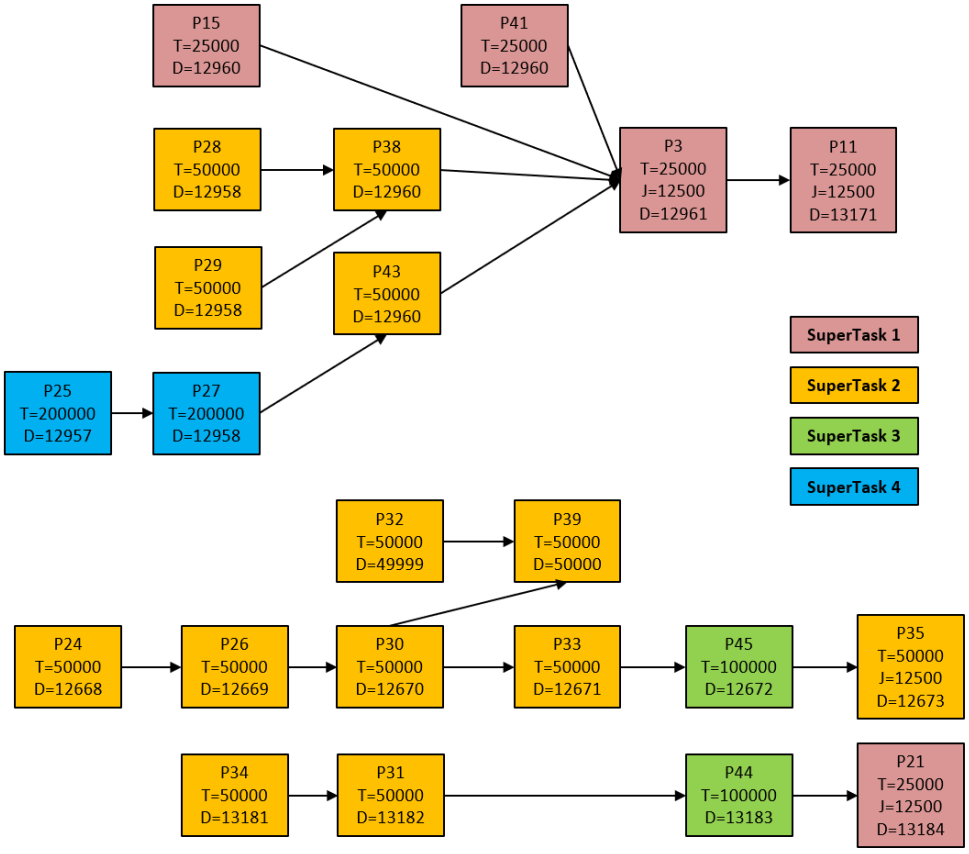


Figure 4.7: Clustering Results From Applying Algorithm 6 to the Open Source Control System Example.

| | #SuperTasks | Schedulable Tasks | Transaction Pass? |
|---|---|---|---|
| NoClustering | 75 | 70.7% | Yes |
| Period | 7 | 62.7% | No |

Table 4.5: Clustering Results When Applied to an Aircraft Engine Control Case Study.

| | $\delta_S$ | $\delta_E$ | $\delta_T$ | $\delta_{SUM}$ |
|---|---|---|---|---|
| NoClustering | 3.8% | 4.6% | 2.5% | 10.9% |
| Period | 0.4% | 0.4% | 1.5% | 2.3% |

Table 4.6: Clustering Results When Applied to an Aircraft Engine Control Case Study.

The results when performing schedulability analysis on the newly clustered SuperTask set based on the open source example is shown is Tables 4.5 and 4.6. The results show that the SuperTask set produces a system with significantly lower overheads, reduced by a factor of five. However, the clustering algorithm fails to fulfill transactional requirements because of a lack of focus on ensuring the correct ordering of transactions. Secondly, the number of tests failing their response time analysis actually increases. This was identified to be down to the failure of tasks with jitter requirements to comply with their temporal requirements as they were prioritised lower down the system schedule.

### 4.4.4 Porting Tasks By Transaction

Clustering tasks by period identified issues where task transactions, or jitter requirements, were not taken into account. Therefore the second approach investigated focused principally on transactional requirements. Additionally, as most tasks with a jitter requirement also form part of a transaction, this method also aimed to capture jitter requirements. The algorithm for the transaction clustering method is shown in Algorithm 7. In this case the tasks in transactions were first broken out into SuperTasks, before the remaining task set was ordered into SuperTasks based on their period. Next, the clustered SuperTask sets are prioritised using the DMPO.

**ALGORITHM 7:** OrderTaskSet(Transaction).

```
1:  /* Iterate over each transaction, in reverse order, to ensure that each task
    preceding another in a transaction has a lower deadline */
2:  while TaskSetChanging do
3:     TaskSetChanging = FALSE
4:     for j ∈ TransactionSet do
5:        for i ∈ Transaction_j do
6:           Add i to OrderTaskSet
7:           if i == Transaction'_j FIRST then
8:              k = i
9:           else if OrderTaskSet[k] follows OrderTaskSet[i] then
10:             OrderTaskSet = SwapTask(i, k)
11:             TaskSetChanging = TRUE
12:             k = i
13:          end if
14:       end for
15:    end for
16: end while
17:
18: /* Iterate over the tasks not yet ordered and order based on Period */
19: while Any Task ∉ OrderTaskSet do
20:    MinPeriod = MAX
21:    for j ∉ OrderTaskSet do
22:       if T_j < MinPeriod then
23:          k = j
24:          MinPeriod = T_j
25:       end if
26:    end for
27:    Append k to OrderTaskSet
28: end while
29: return OrderTaskSet
```

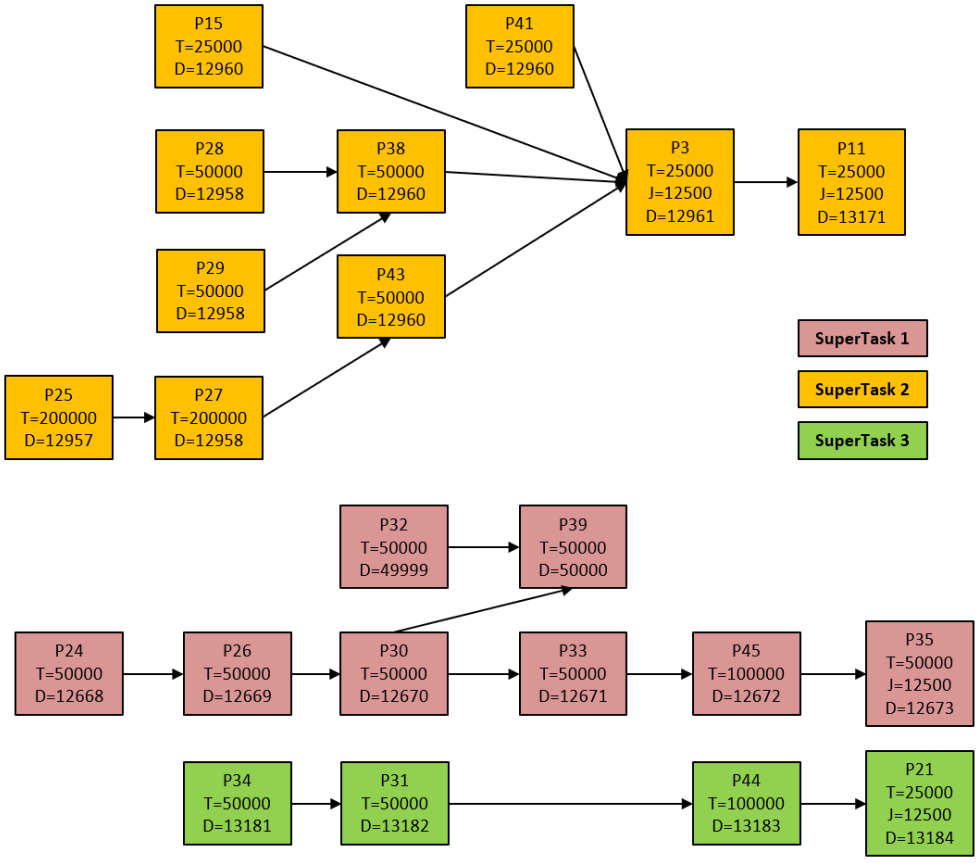


Figure 4.8: Clustering Results From Applying Algorithm 7 to the Open Source Control System Example.

| | #SuperTasks | Schedulable Tasks | Transaction Pass? |
|---|---|---|---|
| NoClustering | 75 | 70.7% | Yes |
| Period | 7 | 62.7% | No |
| Transaction | 4 | 49.3% | Yes |

Table 4.7: Clustering Results When Applied to the Open Source Engine Control Case Study.

| | $\delta_S$ | $\delta_E$ | $\delta_T$ | $\delta_{SUM}$ |
|---|---|---|---|---|
| NoClustering | 3.8% | 4.6% | 2.5% | 10.9% |
| Period | 0.4% | 0.4% | 1.5% | 2.3% |
| Transaction | 2.3% | 2.7% | 2.0% | 7.0% |

Table 4.8: Clustering Results When Applied to the Open Source Engine Control Case Study.

Figure 4.8 shows the SuperTask breakdown for the open source control system. As opposed to the Period clustering method this algorithm ensures that all transactions are maintained inside individual SuperTasks. All tasks that do not form part of a transaction are then broken down into SuperTasks based on their period, before each SuperTask is prioritised using the DMPO.

Tables 4.7 and 4.8 show the results from applying this clustering algorithm to the open source control system. The results confirm that all transactional requirements have been complied with, and again the overhead for the RTOS has been reduced. Interestingly the number of SuperTasks, when compared to the Period clustering technique, is lower; however, the RTOS overhead is higher. This is because while there were only four tasks in the system, these four tasks had very low periods and therefore much higher overheads over a given timeframe.

### 4.4.5 Porting Tasks By Jitter

A second iteration from the Period clustering method was developed to focus on tasks with jitter requirements. This algorithm, denoted in Algorithm 8, prioritises tasks with jitter requirements. Secondly, tasks that form part of a transaction are moved into SuperTasks, in a similar

way to the clustering by transaction algorithm; before all remaining tasks are broken down based on their period.

**ALGORITHM 8:** OrderTaskSet(Jitter).

```
1:  /* Add tasks with jitter requirements into OrderTaskSet in period order */
2:  while TaskSetChanging do
3:      TaskSetChanging = FALSE
4:      for i ∉ OrderTaskSet do
5:          if J_i ≠ 0 then
6:              Add i to OrderTaskSet
7:              for j ∈ OrderTaskSet do
8:                  if j == OrderTaskSet[0] then
9:                      k = j
10:                 else if T_k > T_j then
11:                     OrderTaskSet = SwapTask(j, k)
12:                     TaskSetChanging = TRUE
13:                     k = j
14:                 end if
15:             end for
16:         end if
17:     end for
18: end while
19:
20: /* Use the cluster by Transaction algorithm to sort the remaining tasks. */
21: OrderTaskSet = OrderTaskSet(Transaction)
22: return OrderTaskSet
```

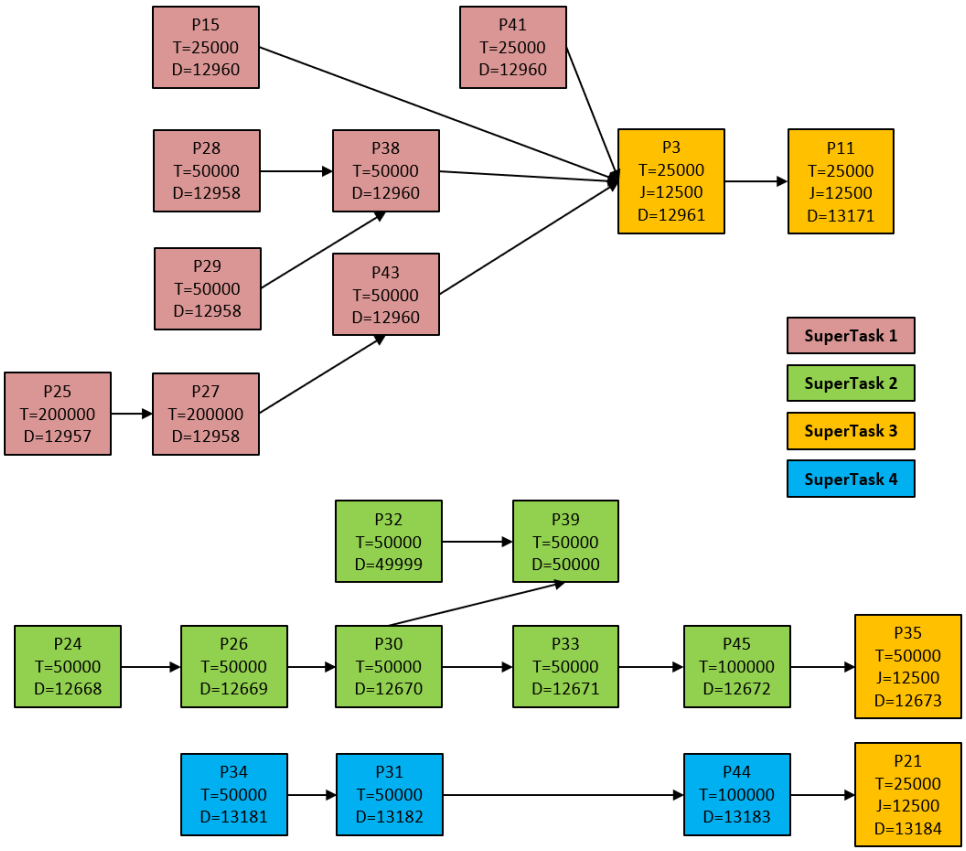


Figure 4.9: Clustering Results From Applying Algorithm 8 to the Open Source Control System Example.

| | #SuperTasks | Schedulable Tasks | Transaction Pass? |
|---|---|---|---|
| NoClustering | 75 | 70.7% | Yes |
| Period | 7 | 62.7% | No |
| Transaction | 4 | 49.3% | Yes |
| Jitter | 14 | 33.3% | No |

Table 4.9: Clustering Results When Applied to the Open Source Engine Control Case Study.

| | $\delta_S$ | $\delta_E$ | $\delta_T$ | $\delta_{SUM}$ |
|---|---|---|---|---|
| NoClustering | 3.8% | 4.6% | 2.5% | 10.9% |
| Period | 0.4% | 0.4% | 1.5% | 2.3% |
| Transaction | 2.3% | 2.7% | 2.0% | 7.0% |
| Jitter | 4.6% | 5.5% | 2.7% | 12.7% |

Table 4.10: Clustering Results When Applied to the Open Source Engine Control Case Study.

Figure 4.9 shows the breakdown of SuperTasks following execution of the cluster by jitter algorithm. Tables 4.9 and 4.10 show the schedulability analysis results obtained from analysing the jitter clustered task set for the open source control system example. Again, the results show a further degradation in the number of schedulable tasks. In this case partly down to the increased number of SuperTasks, with its resultant increase in scheduler overheads.

### 4.4.6 Porting Tasks By Deadline

Clustering tasks by Period, Transaction or Jitter has proved unreliable as focusing on one of these requirements and neglecting others has proved an ineffective way of clustering tasks. This was perceived to be because it is not possible to prioritise one temporal parameter over another, without producing an un-schedulable system. Therefore, the final algorithm focuses on clustering tasks by Deadline. As the Deadline is calculated using the period and any transactional or jitter requirements, this approach should ensure all temporal requirements receive equal weight. The technique is shown in Algorithm 9.

**ALGORITHM 9:** OrderTaskSet(Deadline).

```
1: /* Order tasks by Deadline */
2: OrderTaskSet = UnOrderTaskSet
3: while TaskSetChanging do
4:    TaskSetChanging = FALSE
5:    for i ∈ OrderTaskSet do
6:       for j ∈ (i...τ_j) do
7:          if D_j < D_i then
8:             OrderTaskSet = SwapTasks(i, j)
9:             TaskSetChanging = TRUE
10:         end if
11:      end for
12:   end for
13: end while
14: return OrderTaskSet
```

There are two variants to this algorithm as follows:

- Deadline_D - The algorithm does not allow tasks to co-exist in the same SuperTask if they have different deadlines. This aims to ensure that tasks with tight deadlines can be prioritised accordingly.
- Deadline_P - The algorithm does allow tasks with dissimilar deadline to co-exist within a SuperTask, provided the rules set out in the original rules within Algorithm 4 are maintained.

Figure 4.10 shows the SuperTask breakdown for the Deadline_D clustering algorithm. The Deadline_P equivalent diagram is not shown as all tasks in this set of transactions would be placed within the same SuperTask.

The results from applying the two additional clustering algorithms to the open source control system are shown in Tables 4.11 and 4.12. The results show how the use of each task's deadline as a clustering

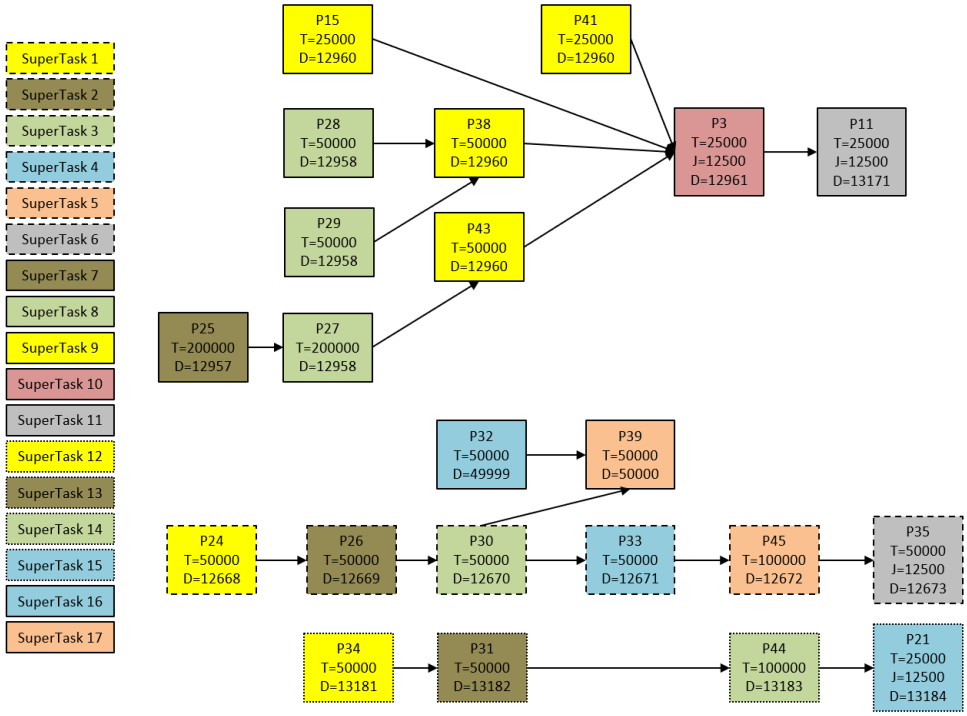


Figure 4.10: Clustering Results From Applying Algorithm 9 to the Open Source Control System Example.

technique produces improved results. In both cases all task transactional requirements are complied with. However, the Deadline_D algorithm produces a system with a larger number of SuperTasks, which produces RTOS overheads so high that the response time analysis of tasks lower down the system schedule fails.

Comparing the results produced across all of the clustering algorithms the results appear to indicate that clustering by deadline is the most appropriate technique. Furthermore, the results show how the Deadline_P algorithm does not produce the system with the lowest number of SuperTasks, or the lowest RTOS overhead. This confirms that using optimisation targets such as lowest number of SuperTask or minimum overheads are not the only objectives for this process.

| | #SuperTasks | Schedulable Tasks | Transaction Pass? |
|---|---|---|---|
| NoClustering | 75 | 70.7% | Yes |
| Period | 7 | 62.7% | No |
| Transaction | 4 | 49.3% | Yes |
| Jitter | 14 | 33.3% | No |
| Deadline_D | 43 | 90.7% | Yes |
| Deadline_P | 10 | 100.0% | Yes |

Table 4.11: Clustering Results When Applied to an Aircraft Engine Control Case Study.

| | $\delta_S$ | $\delta_E$ | $\delta_T$ | $\delta_{SUM}$ |
|---|---|---|---|---|
| NoClustering | 3.8% | 4.6% | 2.5% | 10.9% |
| Period | 0.4% | 0.4% | 1.5% | 2.3% |
| Transaction | 2.3% | 2.7% | 2.0% | 7.0% |
| Jitter | 4.6% | 5.5% | 2.7% | 12.7% |
| Deadline_D | 1.7% | 2.0% | 1.9% | 5.6% |
| Deadline_P | 0.4% | 0.5% | 1.5% | 2.5% |

Table 4.12: Clustering Results When Applied to an Aircraft Engine Control Case Study.

This study has identified a potential clustering method that works well with the industrial case study obtained from Bate [38]. The following sections now perform a wider study of the clustering techniques to investigate whether the method is still the most appropriate when conducted against a large number of different systems.

### 4.4.7 Results from Applying the Clustering Techniques to the Rolls-Royce Control System

The clustering algorithms were first applied to the Rolls-Royce control system introduced in Section 4.3. The results are shown in Tables 4.13 and 4.14.

| | #SuperTasks | Schedulable Tasks | Transaction Pass? |
|---|---|---|---|
| NoClustering | 228 | 24.1% | Yes |
| Period | 17 | 85.5% | No |
| Transaction | 10 | 9.2% | Yes |
| Jitter | 53 | 38.2% | No |
| Deadline_D | 167 | 40.4% | Yes |
| Deadline_P | 15 | 100.0% | Yes |

Table 4.13: Clustering Results When Applied to the Rolls-Royce Aircraft Engine Control System.

| | $\delta_S$ | $\delta_E$ | $\delta_T$ | $\delta_{SUM}$ |
|---|---|---|---|---|
| NoClustering | 17.6% | 21.1% | 6.3% | 45.0% |
| Period | 2.5% | 3.0% | 2.1% | 7.6% |
| Transaction | 4.0% | 4.8% | 2.5% | 11.4% |
| Jitter | 13.1% | 15.7% | 5.1% | 33.9% |
| Deadline_D | 11.7% | 14.1% | 4.7% | 30.5% |
| Deadline_P | 0.8% | 0.9% | 1.6% | 3.3% |

Table 4.14: Clustering Results When Applied to the Rolls-Royce Aircraft Engine Control System.

The results show that the Deadline_P clustering method was the only algorithm able to generate a schedulable system. This was despite the

fact that it was not the algorithm that produced the task set with the smallest number of Super Tasks. The Period and Transaction clustering algorithms failed to prioritise tasks with jitter requirements, and so those tasks presented worst case response times that would have failed to meet their tight timing requirements. Whereas the Jitter clustering algorithm failed to correctly order transactions, and created a system with a larger number of high rate SuperTasks, leading to a higher RTOS utilisation which left the system unschedulable. The Deadline_D method correctly ordered transactional tasks and prioritised tasks with jitter requirements. However, as it did not group together tasks with different deadlines, it created a system with a prohibitively large RTOS overhead.

In comparison to the original system, this partitioned approach allowed low criticality tasks totalling 44% utilisation to be added into the system without compromising schedulability across all clustering algorithms. This would not have been possible in the existing legacy system and was only made feasible as the analysis was able to capitalise on the difference between each high criticality task's $C_i^{LO}$ and $C_i^{HI}$.

### 4.4.8 Large Scale Evaluation

The previous sections have tested the clustering algorithms against two real control systems. However, it is important to ensure that these results are reflected when applied to a large range of different systems to ensure that the algorithms are not biased towards the two control systems tested so far. To ensure the avoidance of bias a number of system parameters were varied through the analysis, these included overhead rates, size and length of transactions and the number of tasks with jitter requirements.

In order to perform this analysis a large number of task sets were randomly generated, with the clustering algorithms being applied to each

set. The random task set generator used is based on a version of the UUniFast algorithm [89], and was extended, as detailed below, to feature jitter requirements and transaction requirements. The random task set generation assessment was performed at varying target utilisations from 30% to 100% (at an interval of every 5%), with a varying number of tasks (10, 50, 100). Each clustering technique was then applied to each generated task set. Finally, the result was statically analysed to confirm every task's response time was less than its deadline and that each transaction was correctly ordered. One thousand tests were then performed for each test configuration. Finally, to provide an ideal case for comparison, a zero overhead test was also performed across each generated task set.

Key characteristics of the real engine control software were identified (and simplified) to constrain the generated tasksets as follows:

- Harmonic periods from the set (2.5, 5, 10, 12.5, 25, 50, 100, 200, 500)ms, inline with the real system introduced in section 4.3.

- 5% of tasks randomly chosen to contain a jitter requirement. If part of a transaction only a task at the beginning or end of the transaction was given a jitter requirement.

- Transactions consisting of three tasks, randomly chosen from the existing set. The number of transactions in the system was set to one fifth of the number of tasks, and transactions could include tasks with different periods.

- The $C_i^{LO}$ for each task was randomly defined based on the system level target utilisation. Each task's $C_i^{HI}$ was randomly selected from the range $C_i^{LO} \leq C_i^{HI} \leq 2C_i^{LO}$.

- The criticality of each task was randomly selected to produce a

system with between 60% and 80% high DAL tasks, with the remaining tasks set to be low DAL tasks.

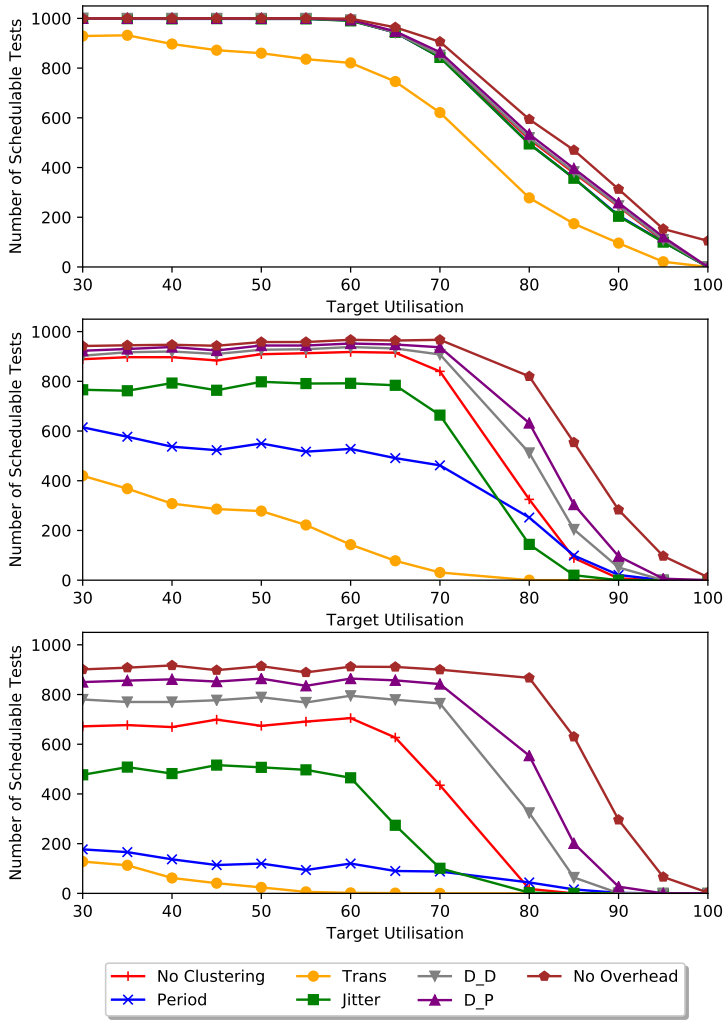


Figure 4.11: Schedulability of a 10, 50 and 100 Task System at Varying Target Utilisations.

This follows principles similar to the approach defined by Kramer et al. [90] where key characteristics are extracted from a real application and fed into a generator to derive representative benchmarks. However, the tasksets used by Kramer et al. [90] follow the AUTOSAR runnable

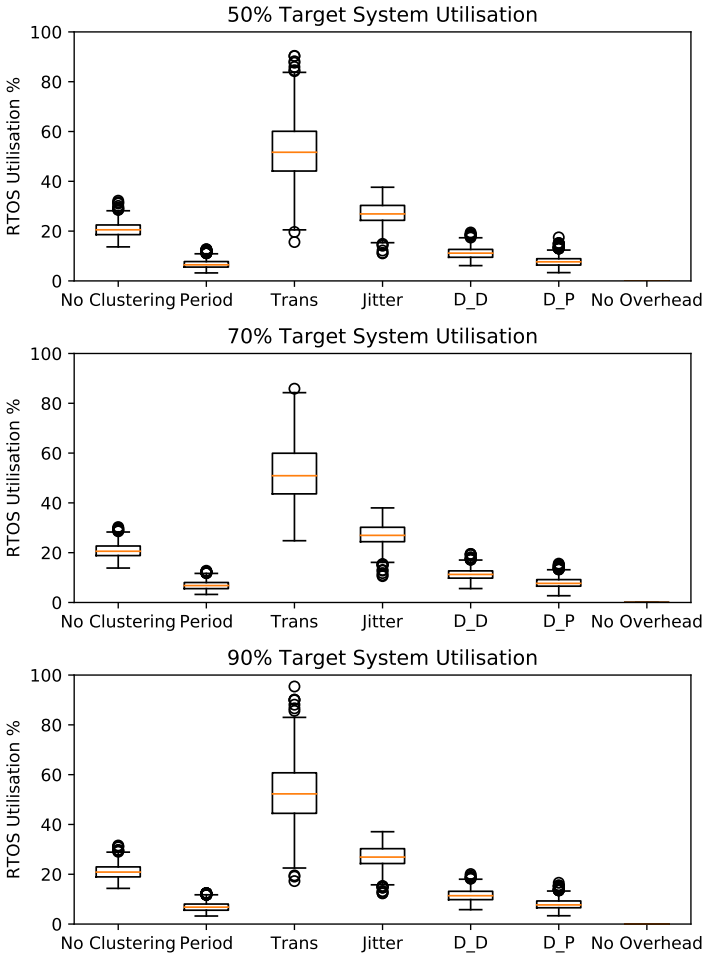


Figure 4.12: RTOS Overheads Calculated for each Clustered System.

model and do not include transactions which have a profound effect on the scheduling approach.

One break from the real engine control system is in the number of tasks defined for each system. In the large scale evaluation task sets of size 10, 50 and 100 tasks are used, whereas the Rolls-Royce system contains more than 200 tasks. Smaller task sets are used because it did not prove possible to define (without bias) a task set containing hundreds of tasks that were schedulable even without system overheads.

Figure 4.11 shows the number of schedulable tests out of the one thou-

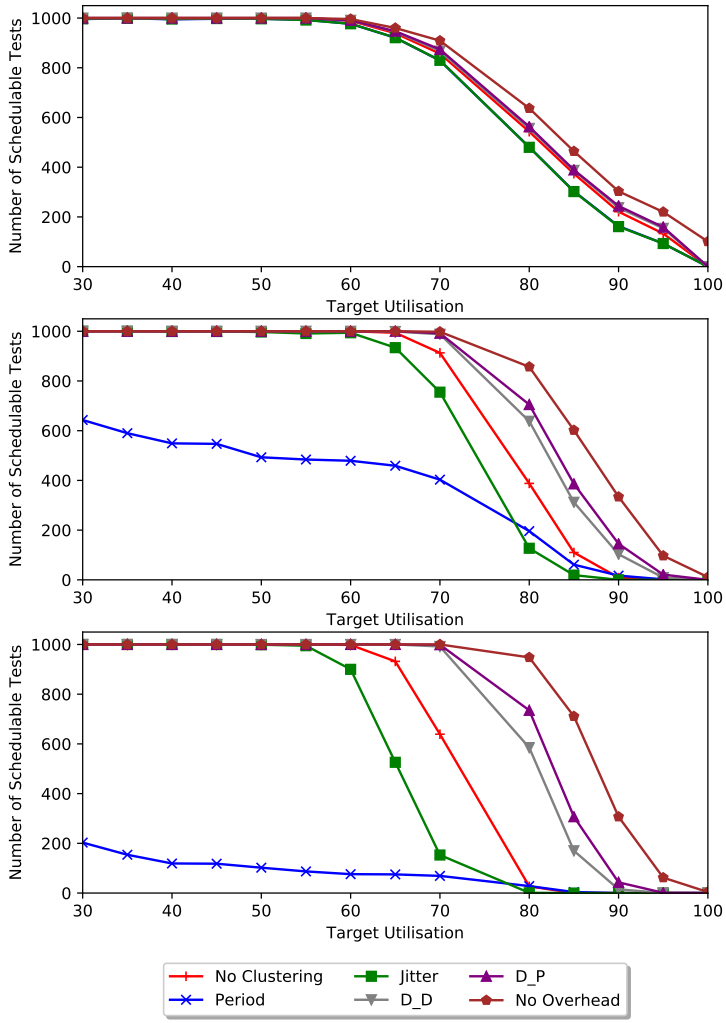


Figure 4.13: Schedulability of a 10, 50 and 100 Task System With No Transactions.

sand executed for each clustering algorithm at varying target utilisation configurations. The experiments showed that for a small task system there was not a great difference across the different methods, with the exception of the *Transaction* method. From the inspection of the results this was largely because the *Transaction* method failed to take account of tasks with tight jitter requirements, which consequently receive lower priorities and failure against their response time analysis.

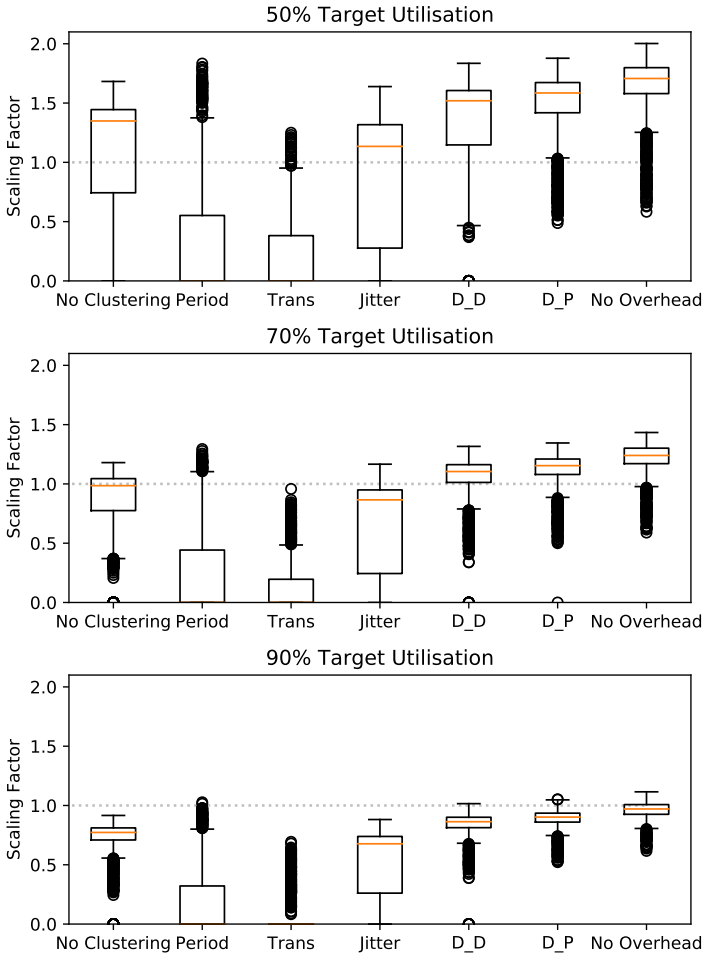


Figure 4.14: Maximum WCET Scaling Factor to Provide a Schedulable System.

For a system with 50 tasks, as shown in the second plot of Figure 4.11, the difference between the clustering methods is more profound. Neither the *Transactions* nor the *Period* methods are able to generate reliably schedulable systems, failing to take account of jitter requirements. The *Jitter* algorithm fares better, but a general failure to preserve transactions causes the schedulability of the solutions to suffer as the task set utilisation grows. The only algorithms able to track near the *No Overhead* ideal are the *Deadline* algorithms, with the *Deadline_P* faring best

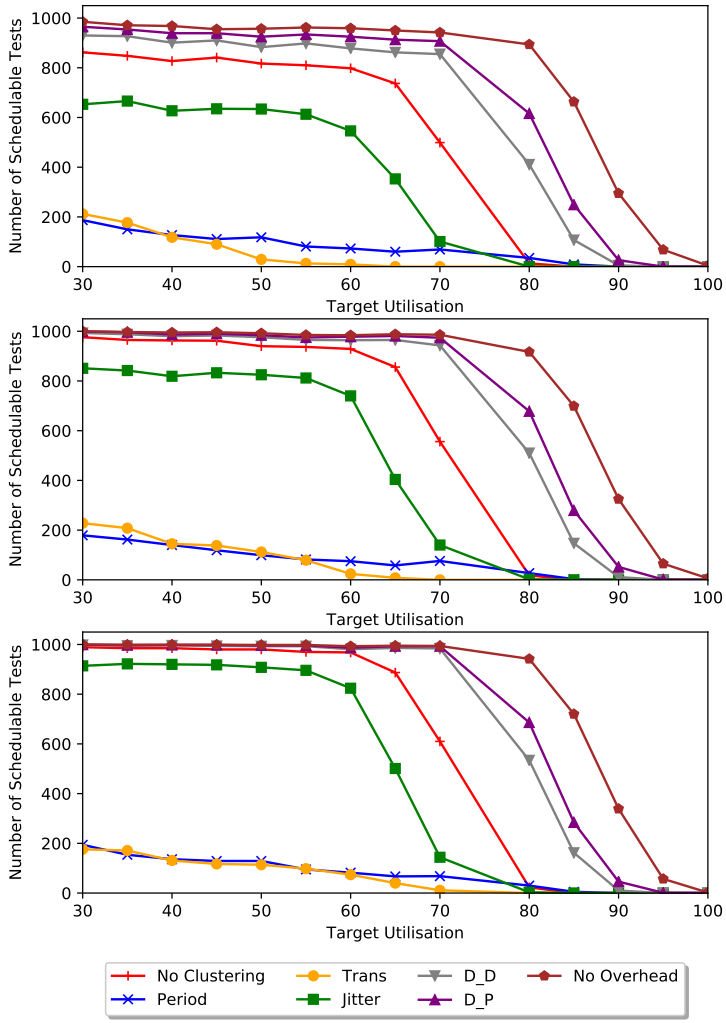


Figure 4.15: Number of Schedulable Tasks with Varying Transaction Rates [10%, 25% and 50%].

as it is able to minimise RTOS overheads by producing systems with less SuperTasks. This hypothesis is further supported in Figure 4.12. This shows the RTOS overhead produced by each clustering method.

These results are amplified as the task set size grows to 100 tasks, where again the only algorithm following a similar trend to the *No Overhead* ideal is the *Deadline_P* algorithm. One irregularity with the results is the fact that for 50 and 100 task systems no clustering algorithms are

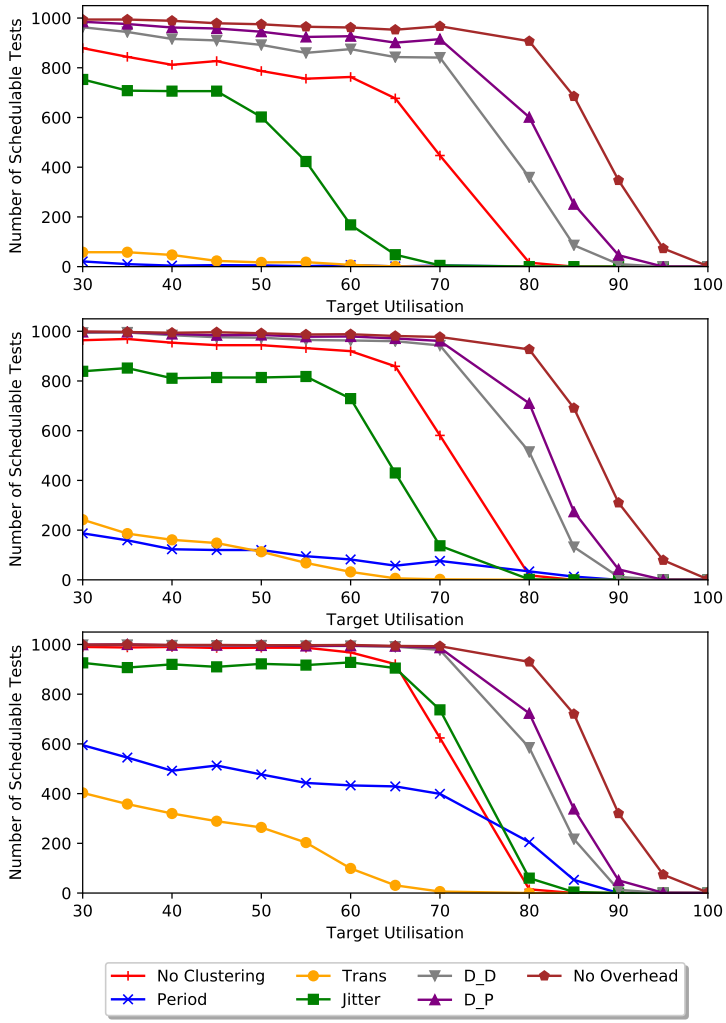


Figure 4.16: Number of Schedulable Tasks with Varying Jitter Rates of [0%, 5% and 10%].

able to achieve a 100% set of schedulable tests. This is because of the effect of transactions as shown by Figure 4.13 which shows the same test as shown by Figure 4.11, however without Transactions.

Figure 4.14 shows the analysed maximum possible WCET inflation factor, or sensitivity, for a 100 task system at varying target utilisation (50%,70%,90%). That is, the maximum figure that every $C_i^{LO}$ and $C_i^{HI}$ can be multiplied by before the system is no longer schedulable. There-

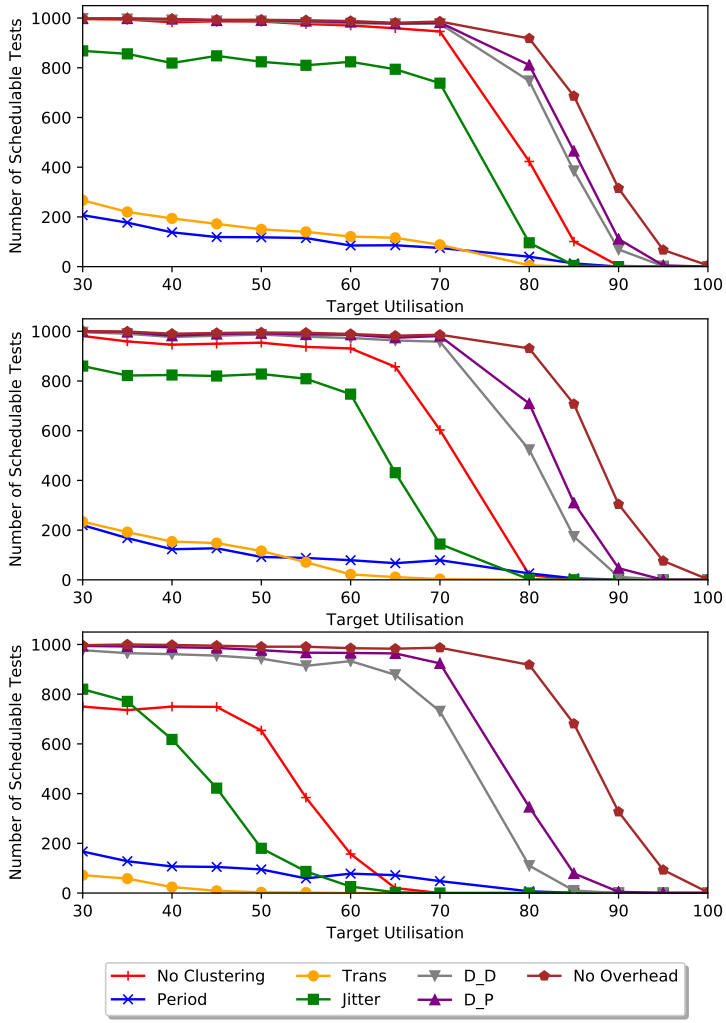


Figure 4.17: Number of Schedulable Tasks with Low, Medium and High RTOS Overheads.

fore, a value above or below one would indicate an increase or decrease (for an initially unschedulable system) in task times respectively. The results showed the *Deadline_P* clustering method maintaining the highest inflation factor across all three target utilisations with other algorithms, in particular *Period* and *Transaction*, tracking inflation factors close to zero. The results further indicate that even *No Clustering* is frequently better than *Jitter*, *Period* and *Transaction*.

Comparing Figure 4.12 to Figures 4.11 and 4.14; even though *Period* tended to have the lowest overhead, it tended to produce less schedulable solutions. This is because the algorithm frequently produces a system with the lowest number of RTOS tasks. However, these tasks do not take account of jitter or temporal requirements, and so is in general not schedulable. Either because tasks with jitter requirements have high response times, or because transaction orders are not maintained. This further supports the assertion that this clustering operation is not necessarily aiming to simply minimise RTOS overheads.

In order to further review the effectiveness of the different clustering algorithms the analysis was extended through application to different systems with varying transactions rates (Figure 4.15), varying jitter rates (Figure 4.16) and varying overheads (Figure 4.17). This analysis shows how the clustering algorithms performed when presented with different system configurations which moved beyond the assumptions introduced by the avionic control system.

Again, the results showed that the *Deadline_P* was reliably the best clustering method. It was shown to be reliable while other clustering algorithms' performance varied significantly across the different system parameters.

## 4.5 Resilient System Design

The previous sections of this chapter have aimed to show how an MCS can be developed based on the AMC protocol. This chapter now explores how this work can be extended to the Resilient model as proposed by Burns et al. [34]. The resilient model aims to apply principles of graceful degradation to the MCS problem by introducing the notion of robust tasks.

This model aligns well to an industrial use case as the model promises to delay moving into the high criticality mode to a later point than potentially offered by the AMC model without dynamic reconfiguration of the system. Furthermore, the concept of a robust task is well aligned to the design of a control system. A robust task is a task that is able to drop one job without affecting its system level requirements [34]; where said task's robustness is independent from its criticality. In the Rolls-Royce control system there exists a number of tasks, or operations, that can be considered as robust. These include for instance, tasks that communicate with monitoring equipment or write tracing data to non volatile memory devices. Both operations could feasibly be paused for a short period of time without having system level safety consequences, provided they are able to restart their operations and execute their operations for a period of time after the pause in service.

So, while the focus of the resilient model is within the field of MCS, it also offers the potential to increase the system utilisation of single criticality systems, provided one or many tasks can be treated as robust.

Studying the scheduler design discussed in Section 4.2; the resilient model can be easily inserted into the model with the only changes made being inside the scheduler around the Handle Overrun operation and with the static schedulability analysis process. These required changes are discussed in the following sections.

### 4.5.1 Handling Overruns

The system state model for the resilient algorithm is shown in Figure 4.18. The operation of the scheduler in each mode is discussed below:

- **Normal Mode**
  - All tasks are released and executed when defined by their temporal requirements.
  - All low criticality tasks are prevented from executing further than their $C_{LO}$.
  - Should any high criticality task execute for longer than its $C_{LO}$ then the Job Failure (JF) is incremented.
  - If the JF counter increases above the Fail Operational (F) threshold, then the system reverts to the Resilient mode.
  - Each time the system reaches the idle task, the JF counter is reset to zero.
- **Resilient Mode**
  - Each robust task skips a number of jobs equal to the task's design time determined skip factor - $S_j$.
  - All other tasks are released, and executed as with the Normal Mode, with the JF count continuing to record high criticality $C_{LO}$ overruns.
  - If the JF counter increases above the Fail Robust (M) threshold, then the system reverts to the High Criticality mode.
  - Should the system reach the idle task, the JF counter is reset to zero and the system reverts back to the Normal mode.
- **High Criticality Mode**
  - Only high criticality tasks are permitted to execute.

– If a robust task has not yet dropped $S_j$ jobs since the original move into the Resilient Mode, then it may continue to do so until $S_j$ is reached.

– Should the system reach the idle task, the JF counter is reset to zero, and the system reverts back to the Normal mode.

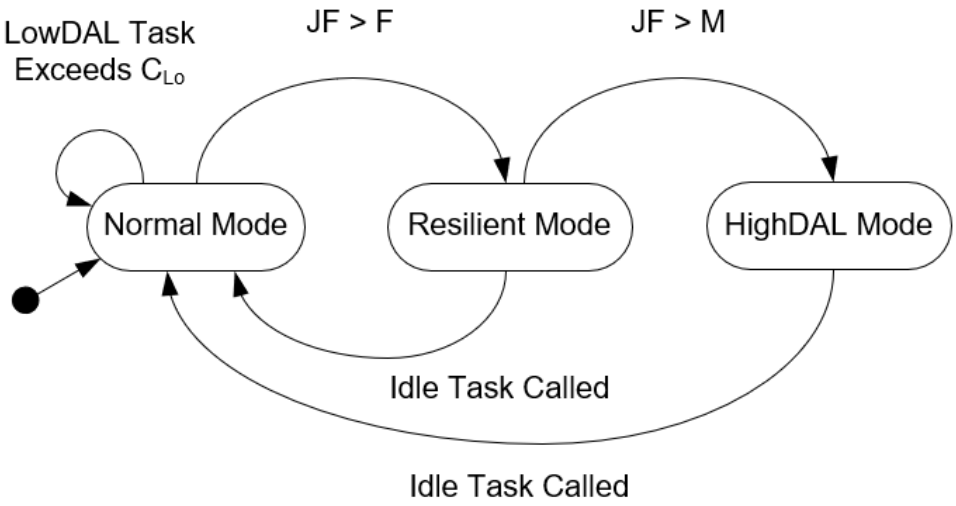


Figure 4.18: Resilient State Flow Diagram.

From this task model it can be assumed that $F < M$, and finally that the same assumptions surrounding the credibility of $C_{LO}$ and $C_{HI}$, as set out in Section 4.2.3, still apply.

This section has explored the operation of the resilient scheduler in each system mode. The following section explores the updates required to the static schedulability analysis of a resilient system to support scheduler overheads.

### 4.5.2 Resilient Schedulability Analysis Updates for Overheads

This section explores the updates required to the static schedulability analysis model presented in [34] in order to support RTOS overheads. This analysis builds off the equations originally defined in Section 4.2.5.

**Normal Mode** - As long as $JF < F$ then all tasks are permitted to execute. However, an additional load must be taken into account to reflect the fact that a number of high criticality tasks (up to $JF$) may execute up to their $C_{HI}$. The original equation as presented in [34] is shown in Equation 4.10.

$$R_i^F = LD(R_i^F, F) + C_i^{LO} + \sum_{j \in hp(i)} \left( \left\lceil \frac{R_i^F}{T_j} \right\rceil C_j^{LO} \right) \tag{4.10}$$

Where LD is a multiset which equals the $F$ largest $C_{DF}[C_{HI} - C_{LO}]$ higher priority task loads. $C_{DF}$ must be added $\left\lceil \frac{R_i^F}{T_j} \right\rceil$ times.

In order to incorporate the overheads identified in Section 4.2.5, this equation must be updated accordingly:

$$R_i^F = LD(R_i^F, F) + C_i^{LO} + C_{START} + \delta_T^F + \sum_{j \in hp(i)} \left( \left\lceil \frac{R_i^F}{T_j} \right\rceil C_j^{LO} \right) + \delta_S^F + \delta_E^F \tag{4.11}$$

**Resilient Mode** - While $F \leq JF < M$ then all tasks are released, however robust tasks will drop up to $S_j$ jobs. The original schedulability analysis equation for the Resilient mode is shown in Equation 4.12. This equation is updated to include overheads in Equation 4.13. In this case $\delta_S$ and $\delta_E$ must be broken down to ensure that the start and stop overheads for robust tasks are correctly handled.

$$R_i^M = LD(R_i^M, M) + C_i^{LO} + \sum_{j \in hp(i)} \left( \left( \left\lceil \frac{R_i^M}{T_j} \right\rceil - S_j \right) (C_j^{LO}) \right) \quad (4.12)$$

Where LD must also be reduced by $(\left\lceil \frac{R_i^M}{T_j} \right\rceil - S_j)$.

$$R_i^M = LD(R_i^M, M) + C_i^{LO} + \delta_T^M + \sum_{j \in hp(i)} \left( \left( \left\lceil \frac{R_i^M}{T_j} \right\rceil - S_j \right) (C_j^{LO} + C_{START} + C_{STOP}) \right) \quad (4.13)$$

This assumes the following:

$$\left\lceil \frac{R_i^M}{T_j} \right\rceil > \left\lceil \frac{R_i^F}{T_j} \right\rceil \quad (4.14)$$

**High Criticality Mode** - Once $JF$ exceeds $M$, the system reverts to the high criticality mode, where only high criticality tasks execute. The analysis of the mode change to this high criticality mode is shown in Equation 4.15

$$R_i^{HI*} = C_i^{HI} + \sum_{j \in hpH(i)} \left( \left( \left\lceil \frac{R_i^{HI*}}{T_j} \right\rceil - S_j^H \right) C_j^{HI} \right) + \sum_{k \in hpL(i)} \left( \left( \left\lceil \frac{R_i^M}{T_k} \right\rceil - S_k^L \right) C_k^{LO} \right) \quad (4.15)$$

This equation can be extended to encompass overheads as follows:

$$R_i^{HI*} = C_i^{HI} + \delta_T^{HI} + \sum_{j \in hpH(i)} \left( \left( \left\lceil \frac{R_i^{HI*}}{T_j} \right\rceil - S_j^H \right) (C_j^{HI} + C_{START} + C_{STOP}) \right) + \sum_{k \in hpL(i)} \left( \left( \left\lceil \frac{R_i^M}{T_k} \right\rceil - S_k^L \right) (C_j^{LO} + C_{START} + C_{STOP}) \right) \quad (4.16)$$

This section has explored how the industrial MCS architecture designed in Section 4.2 can be extended to support a resilient system design. The following section explores whether the process of clustering tasks to support efficient system design is still appropriate for this new scheduler design.

### 4.5.3 Porting an Existing System to the Resilient Model

Section 4.4.2 defined the following rules for breaking a task set down into SuperTasks:

- No SuperTask can contain two tasks of different criticalities. Thus ensuring the MCS rules surrounding partitioning can be maintained at the RTOS level.
- The response time of each individual task must be less than the task's calculated deadline. The deadline being calculated as defined in Algorithm 3.
- For each transaction, any task preceding another task should have a higher priority than the succeeding task.

In order to support a resilient task model the following rule must also be defined:

- No SuperTask can contain two tasks with different robustness skip factors ($S_j$). The SuperTask takes on the robustness skip factor value of its set of tasks.

This rule ensures that the RTOS and scheduler is able to correctly control the release of robust tasks.

The following sections now explore the ability of the clustering algorithms defined in Section 4.4 to porting the existing systems defined in Section 4.3 to a resilient system. The section then continues to explore the clustering algorithm's applicability to a resilient system by using the same random task set generation technique defined in Section 4.4.8.

As with the previous analysis the overhead figures were measured using the qualified Rolls-Royce process, and were defined as follows:

- $C_{TICK} = 37\mu s$
- $C_{REL} = 8\mu s$
- $C_{START} = 28\mu s$
- $C_{STOP} = 31\mu s$

Throughout this section the number of high criticality overruns the system must tolerate (F) is set to 10, and the number of high criticality overruns that must be tolerated while robust tasks skip a job (M) is set to 16. In practice such figures should be chosen to maximise the efficiency and schedulability of the system under development. However, for the purpose of providing a level playing field for the analysis that follows these values were fixed.

### 4.5.4 Open Source Control System

The open source control system taskset was adopted directly from the system introduced in Table A.1, however all low criticality tasks were also treated as robust tasks.

The results from applying the different clustering algorithms to the open source control system are shown in Tables 4.15 and 4.16. The results show that for the resilient system the Deadline_P clustering technique is

| | #SuperTasks | Schedulable Tasks | Transaction Pass? |
|---|---|---|---|
| NoClustering | 75 | 66.7% | Yes |
| Period | 7 | 62.7% | No |
| Transaction | 4 | 49.3% | Yes |
| Jitter | 14 | 33.3% | No |
| Deadline_D | 43 | 88.0% | Yes |
| Deadline_P | 10 | 100.0% | Yes |

Table 4.15: Clustering Results When Applied to the Open Aircraft Engine Control Resilient Case Study.

| | $\delta_S$ | $\delta_E$ | $\delta_T$ | $\delta_{SUM}$ |
|---|---|---|---|---|
| NoClustering | 4.3% | 4.8% | 2.7% | 11.8% |
| Period | 0.4% | 0.4% | 1.6% | 2.5% |
| Transaction | 2.5% | 2.8% | 2.2% | 7.6% |
| Jitter | 5.1% | 5.6% | 2.9% | 13.7% |
| Deadline_D | 1.9% | 2.1% | 2.0% | 6.0% |
| Deadline_P | 0.5% | 0.5% | 1.6% | 2.7% |

Table 4.16: Clustered Overheads When Applied to the Open Aircraft Engine Control Resilient Case Study.

again the most appropriate, being the only algorithm able to produce a schedulable system. In comparison to the same system results obtained for the AMC system, it can be noted that the overheads increased by a small amount, which can be directly attributed to the higher RTOS execution time ($C_{TICK}$, $C_{REL}$, $C_{START}$ and $C_{STOP}$). This increased overhead is induced due to the small additional task release and supervision time that is required to be performed by the RTOS.

The results reflect the fact that the same clustered task set is produced for both the AMC and for the resilient scheduling models. This is because under the AMC model low criticality tasks were broken out into separate tasks. Now as these same tasks are treated as robust they are still moved into their own super tasks in the same way. That is, the addition of the new clustering rule has no effect on this system.

### 4.5.5 Rolls-Royce Control System

| | #SuperTasks | Schedulable Tasks | Transaction Pass? |
|---|---|---|---|
| NoClustering | 228 | 14.5% | Yes |
| Period | 17 | 79.4% | No |
| Transaction | 10 | 7.9% | Yes |
| Jitter | 53 | 31.6% | No |
| Deadline_D | 171 | 33.3% | Yes |
| Deadline_P | 15 | 100.0% | Yes |

Table 4.17: Clustering Results When Applied to the Rolls-Royce Resilient Case Study - Experiment 1.

For the Rolls-Royce control system; two approaches were followed. For the first experiment the whole set of low criticality tasks were treated as robust. The second was a more ambitious approach where as well as low criticality tasks, a number of high criticality tasks which could be identified as providing monitoring output, communication, or non volatile memory accesses were all marked as robust. This produced a system with approximately 10% of tasks being treated as robust.

As with the open source control system example in the previous section; Experiment 1 (results shown in Table 4.17) produced identical re-

| | $\delta_S$ | $\delta_E$ | $\delta_T$ | $\delta_{SUM}$ |
|---|---|---|---|---|
| NoClustering | 19.7% | 21.8% | 7.1% | 48.6% |
| Period | 2.8% | 3.1% | 2.3% | 8.2% |
| Transaction | 4.5% | 5.0% | 2.8% | 12.3% |
| Jitter | 14.7% | 16.3% | 5.7% | 36.6% |
| Deadline_D | 13.4% | 14.8% | 5.3% | 33.4% |
| Deadline_P | 0.9% | 1.0% | 1.7% | 3.6% |

Table 4.18: Clustered Overheads When Applied to the Rolls-Royce Resilient Case Study - Experiment 1.

sults to the same system when ported to the AMC model. Once again this is because robust tasks are treated by the clustering algorithm in the same way as low DAL tasks. When reviewing the schedulability of the system it can be seen that the Deadline_P algorithm is the only clustering algorithm able to produce a schedulable system. Given the clustered system properties for the resilient system, match those of the AMC system, these results are as expected.

Table 4.18 shows the RTOS overheads measured for the robust system once ported to the resilient model. Again in comparison to the AMC model the overheads have increased slightly. This is to be expected given the small increase in required RTOS compute time.

The results for the second experimental Rolls-Royce system are shown in Table 4.19. Before porting this system to the resilient scheduling model a number of tasks were tagged as robust tasks. These tasks were identified as robust based on the effect on their operation and the wider system, if they were to skip a job. This includes, for instance, long-delay memory accessing tasks or communications tasks.

The clustering results indicate that the number of SuperTasks in each

clustered system increased for each clustering method, except for the Transaction based method. This is mirrored with an equivalent increase in RTOS overheads as shown in Table 4.20. The increase in the number of clustered tasks can be attributed to the additional robust tasks which were broken out into new SuperTasks. In the case of the Transaction clustering method - as each transaction exists to regulate and prove data interactions (according to strict timing requirements), no tasks that form part of a transaction could be considered as robust. This means the Transaction clustering method for the resilient system mirrors the operation of the AMC system.

| | #SuperTasks | Schedulable Tasks | Transaction Pass? |
|---|---|---|---|
| NoClustering | 228 | 15.8% | Yes |
| Period | 26 | 82.0% | No |
| Transaction | 10 | 7.9% | Yes |
| Jitter | 61 | 31.1% | No |
| Deadline_D | 172 | 33.3% | Yes |
| Deadline_P | 25 | 100.0% | Yes |

Table 4.19: Clustering Results When Applied to the Rolls-Royce Resilient Case Study - Experiment 2.

As with the AMC clustered results, the results for the clustered Resilient model indicated that only the Deadline_P clustering algorithm was able to produce a schedulable system. These results, combined with the results from the open source aircraft engine control system example in Section 4.5.4, indicate that the developed clustering algorithms are also applicable to the resilient system scheduler. Section 4.5.6 now reapplies the large scale evaluation to the resilient scheduling model to identify whether the clustering approaches are still applicable.

| | $\delta_S$ | $\delta_E$ | $\delta_T$ | $\delta_{SUM}$ |
|---|---|---|---|---|
| NoClustering | 19.7% | 21.8% | 7.1% | 48.6% |
| Period | 3.3% | 3.7% | 2.4% | 9.4% |
| Transaction | 4.5% | 5.0% | 2.8% | 12.3% |
| Jitter | 15.2% | 16.8% | 5.8% | 37.8% |
| Deadline_D | 13.4% | 14.8% | 5.3% | 33.6% |
| Deadline_P | 1.5% | 1.7% | 1.9% | 5.1% |

Table 4.20: Clustered Overheads When Applied to the Rolls-Royce Resilient Case Study - Experiment 2.

### 4.5.6 Large Scale Evaluation

In order to further study the applicability of the clustering algorithms introduced throughout this section, the large scale evaluation originally introduced in Section 4.4.8 was reapplied to the resilient scheduling model.

The task set generator was configured using the same process followed in Section 4.4.8, with the additional step that between 10% and 30% of tasks were randomly identified as robust tasks. Each task's robustness and criticality were treated as independent task parameters.

Figure 4.19 shows the number of schedulable task sets produced for each clustering algorithm when trying to define a 10, 50 and 100 task system. The results correlate with the AMC system results, and again show the Deadline clustering algorithms (in particular the Deadline_P algorithm) to be the clustering technique able to obtain the highest number of schedulable task sets reliably. Overall it can also be noted that the number of schedulable task sets has reduced compared to the AMC algorithm, as indicated by the real system application in the previous section, this can be attributed to the addition of extra SuperTasks to support ro-

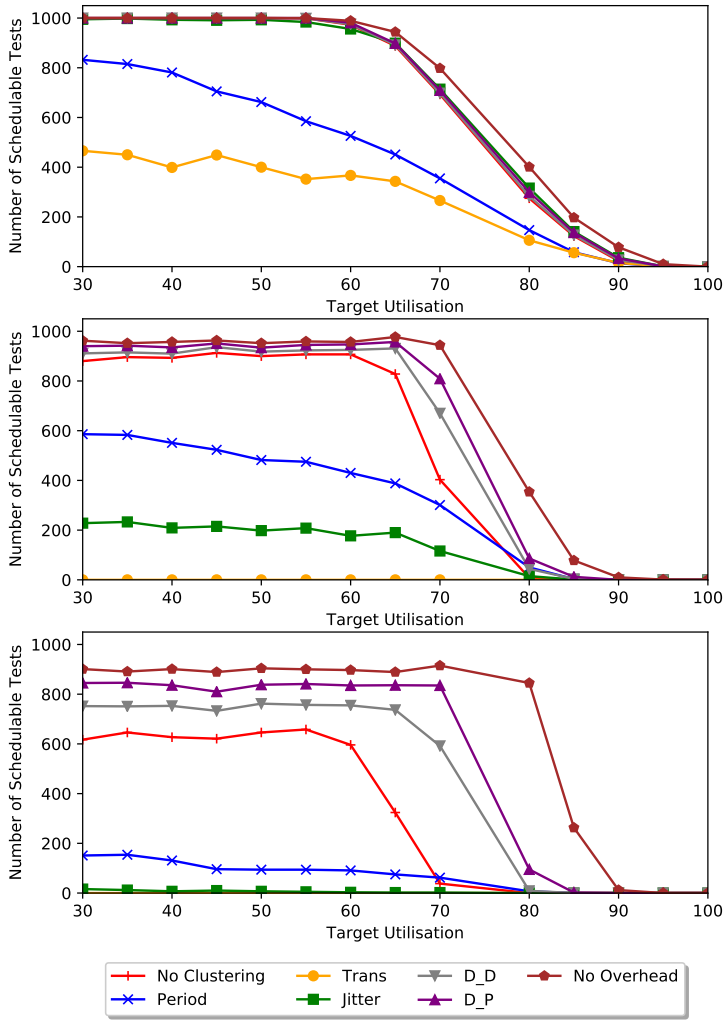


Figure 4.19: Schedulability of a 10, 50 and 100 Task System at Varying Target Utilisations.

bust tasks. The result of which leads to higher RTOS overheads in the produced system.

The results also indicate that the Transaction, Jitter the Period clustering algorithms are further disadvantaged by the introduction of robust tasks, in all cases the algorithms fare worse than a system without clustering at all.

This section has reapplied the clustering algorithms (designed to port

an existing non-preemptive system to the AMC pre-emptive system) to the resilient scheduling model. The aim of the analysis was to provide confidence that the developed techniques are not unique to, or influenced by, the AMC system. The results indicate that the approach is also applicable to the resilient scheduling model, with comparative results between the two systems being obtained.

### 4.5.7 Summary

This chapter has aimed to present the development of a MCS to support industrial applications. The section has discussed the certification requirements for a MCS system and presented a system design aimed at complying with such requirements. The developed system utilises temporal and spatial partitioning to achieve separation between different criticality tasks.

The chapter progressed to discuss how the suggested system could be developed, including the appropriate assessment of task timing properties (using the work discussed in Chapter 3) and the analysis of RTOS overheads. These RTOS overheads are incorporated into an updated set of static schedulability algorithms. The chapter progressed to assess the developed system against the certification requirements defined; this review identifies assumptions and key requirements placed on the system that allows an initial certification safety case to be developed.

Following this system definition the chapter advances to analyse how an existing system may be ported to the new MCS system. This analysis, when faced with unacceptable system overheads, identifies an approach for reducing system overheads through task clustering. Different clustering techniques are identified according to various task timing properties, and a comprehensive assessment is presented which identifies the most

appropriate method. Finally, the system is advanced and extended to support a resilient scheduling model, in order to assess the approaches wider applicability.

Ultimately this chapter has presented a mixed criticality system definition, together with an initial approach to support certification. The newly developed system allows expansion of a Rolls-Royce example system to support additional tasks totalling 40% extra utilisation, which would not have been possible previously.

The approach, which utilises the timing analysis techniques discussed in Chapter 3, provides a method for developing a system, and for proving compliance of high criticality tasks against their requirements. However, the approach lacks the ability to demonstrate adherence of low criticality, or robust, tasks against their temporal requirements. This is because both scheduling methodologies discussed (the AMC+ and Resilient models) rely on the ability to deny service to low DAL, or robust tasks, at certain times. There is currently no process for identifying the length or frequency of such denied service, and therefore no ability to certify, or prove, the system as a whole.

Chapter 5 now expands on this shortfall and discusses an approach aiming to allow low criticality, and/or robust, task certification.

# Chapter 5

# Assessing Low Criticality Task Service

Chapter 3 developed a process for identifying the timing properties of a software task and Chapter 4 built on this work by focusing on the development processes for a Mixed Criticality System (MCS). Together these two chapters provided a process for developing and statically proving the operation of high DAL tasks within an MCS. This provides the mechanisms required for proving compliance of high DAL tasks against their timing requirements. However, this work currently does not provide any process for defining the service provided to low DAL tasks. This chapter now seeks to fill this gap by exploring how a system integrator may assess such service.

Regardless of the scheduling methodology employed, the general assumption in the literature is that low DAL components can be denied service at times of heightened system utilisation. In practice, in a well designed system this should only occur in extreme cases, if ever. Unfortunately, however, this potential lack of service cannot be quantified without analysing the performance of the integrated system in operation.

This is because it is not known how many tasks, if any, may execute beyond their timing bound within a certain time window without executing the system in a representative environment. This means it is difficult to obtain concrete proof that a low DAL component will receive a good enough level of service to fulfil its mission requirements.

One strategy to combat this in an MCS may be to increase $C^{LO}$ timing budgets across the system in order to ensure that low DAL components are 'never' denied service. However, the more a task's $C^{LO}$ increases towards its $C^{HI}$; the less the system is able to benefit from utilising the $C^{HI}$ pessimism. Furthermore, (unless $C^{LO} == C^{HI}$) then even this approach cannot be *guaranteed* to provide the necessary service in *all* cases. The key question to be answered is; how often do tasks exceed their $C^{LO}$ timing budget? In essence, it is difficult to understand the performance afforded to low DAL tasks in an MCS without performing a dynamic assessment in a representative environment.

Several papers have explored approaches aiming to improve low DAL task support. For example, Jan et al. [28] and Su & Zu [32], looked at applying the elastic task model, originally proposed by Buttazzo et al. [29], to an MCS. Rather than de-scheduling all tasks, this model instead extends the period of low DAL tasks to reduce the utilisation on the system, this has also been extended to support graceful degradation by Gill et al. [33]. In contrast, the so-called imprecise mixed criticality model [30] reduces low DAL execution budgets in order to improve wider system performance. This was advanced by Huang et al. [91] with extensions to support graceful degradation. Additionally, Burns & Baruah [92] reduce the priority of low DAL tasks as required, effectively executing low DAL tasks during periods of high system utilisation in system slack time only.

The overriding assumption with all of these approaches is that the

temporal properties of low DAL tasks can be altered, be it by reducing service, or execution time, without affecting its overall requirements. In some cases this may be acceptable. However, this is wholly dependant on the requirements of the low DAL task, which may require real time operation.

Chapter 4 introduced the AMC [25] and resilient models [34], as well as other variants derived from the original Vestal [24] model such as the Bailout model [27]. All three techniques aim to delay a move to the high criticality mode until necessary. However, even though the move to the high DAL mode may be delayed, it still may occur at some point. The proposed protocols do not offer a way to understand when a move may occur, or what impact (by way of duration and frequency of loss of service) this may have on the low DAL, and/or robust, task.

In single criticality systems a number of approaches applicable to ensuring soft real time task service could also be applied to an MCS. Systems exploring tasks with m-k (or m-n) firm deadlines [93], [94] provide a way of formalising the requirements for soft real time (or low criticality) tasks. The methods suggest ways of prioritising tasks that are approaching a failure in their m-k requirement, either through dynamic methods [93] or static methods [94]. However, the techniques still represent a best effort approach, even once tasks approaching m-k requirement errors are offered greater service, an indication of the quality of service that may be achieved is not provided.

Weakly hard systems [95] build on similar principals to m-k-firm systems and offer methods where soft real time tasks can be disabled for defined periods of time. This type of system has already been extended to support an MCS based on the AMC model [96]. In this instance, low criticality tasks are still executed when in the high criticality mode, but

with reduced service as the tasks are forced to skip a statically defined number of jobs. This provides a system integrator with the ability to bound the minimum quality of service afforded to low criticality tasks. The disadvantage with this approach is that the schedulability analysis in the high criticality mode must account for a number of low criticality jobs executing, thereby reducing the potential of an MCS to harness WCET pessimism through exploitation of the difference between $C^{LO}$ and $C^{HI}$.

One paper that has attempted to quantify low DAL service is Medina et al. [97] which defines a probabilistic process for assessing the performance of low criticality tasks. This work offers a good example of how low DAL service could be quantified. However, the paper assumes that task timing error rates are already known, and does not provide guidance on how they could be assessed.

In summary, the predominant MCS models available in the literature concentrate on high DAL task requirements, with the static analysis showing that in the worst case low DAL tasks will receive no service. Methods such as expanding low DAL task execution parameters aimed to improve low DAL task overall service; and the resilient model's use of graceful degradation aimed to provide some control on reducing low DAL task service. However, ultimately, although the previous work offers many ways to help improve low-DAL task performance, it does not address how to assess this performance and service in a meaningful way that could be used to support certification.

From an industrial point of view this represents a significant barrier to the adoption of MCS models in a safety critical context. Should the low DAL components being integrated still carry some safety related consequence of failure, for example DAL C components, then the task's

compliance to its requirements will still need to be assessed. Even if the component's failure cannot affect the safety of the aircraft, in the case of a DAL E component, then it cannot be assumed the component can simply be denied service permanently. The component's operation must have some useful, mission critical, operation; otherwise it would not have been developed in the first place. Therefore, it is essential that system integrators have a process for quantifying the service afforded to low DAL components. If not for safety, then for business or mission critical reasons.

This chapter examines how a system integrator may assess the service afforded to a low DAL task. The aim is to describe a strategy that could be employed to assess the performance given to a low DAL component in order to allow an informed decision on system performance to be made. The process developed, as well as the processes and tooling developed through Chapters 3 and 4 are then applied to a use case taken from a Rolls-Royce aircraft engine control system.

## 5.1 Assessing the Service Afforded to a Low DAL Task

Regardless of the method chosen to control low DAL or robust tasks the performance of said tasks is wholly dependant on the actual performance of the system. Therefore, the process conducted here is based on a statistical assessment of a set of execution results extracted from either a test rig execution during a system-level test campaign, or from a scheduler simulation of the system in question. This chapter predominately follows the results obtained from simulation; the use of which allows a significantly larger data set to be compiled. However, ultimately these results

would be supplemented and improved by results obtained from a full end to end test campaign when the system is available and integrated. The statistical assessment would then be repeated on this new results base.

The simulator is designed to be initialised using execution profiles extracted from the system during task-level testing designed to mimic system behaviour while in operation [37]. This is in line with the process introduced in Chapter 3. This ensures that the execution profiles provide a realistic representation of the task's actual performance when in operation. As an added benefit, the simulation and system provides the facilities required to perform a 'what-if' analysis based on changing error rates.

The results are then input into a statistical assessment that provides a measure of confidence in compliance to low DAL component requirements, as well as providing an understanding of the probability of failing to comply. Together these results should allow a system integrator to make a guided decision on whether the low DAL component's performance is acceptable or not.

The following section introduces a Goal Structuring Notation (GSN) [35] argument for the approach, as well as providing results from applying the analysis to an industrial case study integrated into the AMC+ and resilient models.

### 5.1.1 Goal Structuring Notation

Within this and the following sections, the process is defined using GSN [35]. GSN is chosen to present the process introduced in this section as it represents a widely used and accepted procedure for presenting certification cases. The principal purpose of a goal structure is to show how goals (claims about the system) are successively broken down into sub-

goals until a point is reached where claims can be supported by direct reference to available evidence (solutions). As part of this decomposition, using the GSN it is also possible to make clear the argument strategies adopted (e.g. adopting a quantitative or qualitative approach), the rationale for the approach (assumptions, justifications) and the context in which goals are stated (e.g. the system scope or the assumed operational role). The GSN arguments in this section use Goals (G), Assumptions (A), Statements (St) and Solutions (S) [35].

Figure 5.1 shows the principal GSN argument for the approach. The principal goal (G0) that the service provided to each low DAL task is sufficient, is analysed using a statistical analysis of results obtained by a simulation of the system.

The strategy for the analysis is broken down into four key sub goals, as follows:

- G1 - Specification - The requirement for the low DAL task can be expressed in a form that allows assessment.
- G2 - Evaluation - The simulation output provides an understanding of the likelihood of an error.
- G3 - Confidence - The simulation has obtained sufficient coverage of the system.
- G4 - Validation - The simulation is a valid representation of a real system.

Ultimately these goals are designed to form a circular argument. Should the service afforded to each low criticality task prove to be unacceptable, when assessed against G1, then steps G2 and G3 are designed to be easily repeatable to allow efficient system update. Given that the

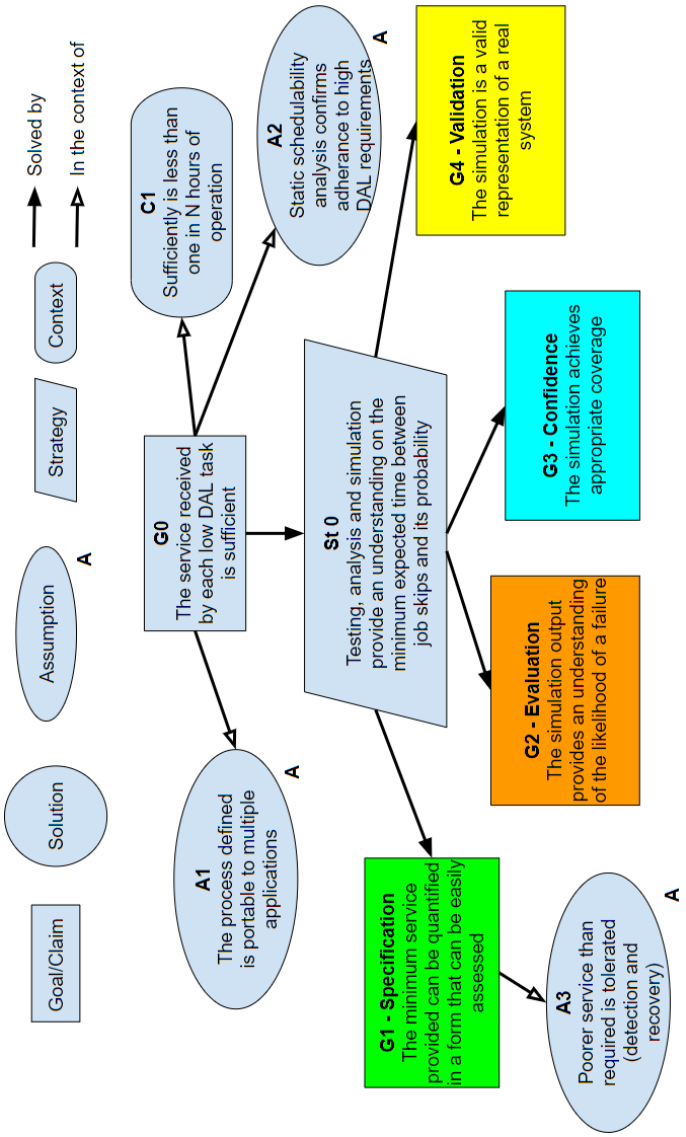


Figure 5.1: Goal Structured Notation Argument for the Overall Low DAL Requirement.

process is built around a representative simulation of the system (as assessed by G4), this repetitive process can be performed at an early, cost effective point in the design life-cycle.

To support this process the following assumptions have been made about the wider system:

- The occurrence of an individual task overrun is very rare. *Rationale: The defined $C^{LO}$ for each task, representing the computation time beyond which a task would register a fault, has been generated from an extensive testing regime and carries with it a high level of confidence. However, being derived from a simple measurement technique it should still be assumed to be optimistic. Very rare assumes to be less than 1 in 10,000 task invocations*

- Individual task overruns are independent and are not reliant on the current operation of the control system. *Rationale: an overrun is an event unique to each task, and not a systematic event caused by an operation at the system level. For example, response to an engine fault condition.*

- Task overruns can be assumed to be independent of hardware operation. *Rationale: The system is designed to be resilient to external hardware failures. Secondly, the target processor design is compliant with DO-254 recommendations as a high criticality device, and has been designed to be resilient against sources of error, such as Single Event Upset.*

The following sections now describe each of these goals in more detail.

### 5.1.2 Specification

In order to assess the service required for a low DAL, or robust, task to execute against its requirements, it is important to specify the problem appropriately. The first requirement for a task to be considered as low DAL is that the task must be able to skip one, or even all, of its jobs without affecting the safety of the overall system.

For a task to be considered robust, but high DAL, in the resilient model; it is necessary for the task to be able to miss $S$ jobs, with $S$ being the task's job skip parameter.

The process proposed in this thesis uses a scheduler simulator. The simulator takes as input a set of timing profiles for each task, generated through the process identified in Chapter 3. The simulator acts as the scheduler, randomly selecting an execution for the highest priority ready task from the task's timing profile. The simulator then advances a record of the current time to the lower of either the next task release time, or to the completion time of the highest priority task.

Using this simulator (which executes approximately 10 times faster than real processor time) repeated combinations of task execution times can be examined to measure the service received by each low DAL task. The simulator outputs the time measured between each individual job skip burst. A job skip burst is defined as a successive set of job skips, that does not stretch over the idle task.

Each low DAL, or robust task, requirement should then define two parameters to allow requirement testing:

- The maximum permissible length of a job skip burst. In other words, the maximum allowed number of jobs that can be skipped consecutively.

- The minimum time between each job skip burst.

Crucially, the definition of the required job skip intervals should be accompanied by a safety assessment which identifies a chance or frequency, of failure that may be deemed acceptable. A failure being when the task fails to comply with its requirements. This level of failure can then be used throughout the Evaluation (discussed in the following sec-

tion) to support an engineering judgement on what may or may not be acceptable.

### 5.1.3 Evaluation

Once the specification to be assessed has been properly defined, the next step in the process is to analyse the results obtained by the scheduler simulator to understand the probability of failing to comply with the low DAL task's requirements. This aims to provide a real measure that can be used to make a decision on whether the service given to the low DAL task is acceptable or not.

Figure 5.2 shows the process for understanding the probability of the low DAL task suffering a timing requirement error. Goal G2 is split into two parts. The first is an assessment based on the observed performance of the system (G5, G6), and the second is a statistical inference to understand the exceedance probability of the sample (G7).

***Are we confident the requirement will be complied with?***

Goals G5 and G6 examine the full range of results obtained from the simulator. The target is to review whether the spread of job skip interval times provide confidence that requirement will be complied with in a significant majority of cases.

Goal G5 advocates the use of box plot diagrams to allow a visual assessment of the median and inter-quartile range of job skip times. To provide confidence it should be confirmed that any times close to the requirement are outliers, and do not represent a substantial percentage of the results obtained from the simulator.

Secondly, Goal G6 uses a percentile test to provide a statistical measure indicating where the majority of results reside. Again any times close to the requirement should be confirmed as in the minority.

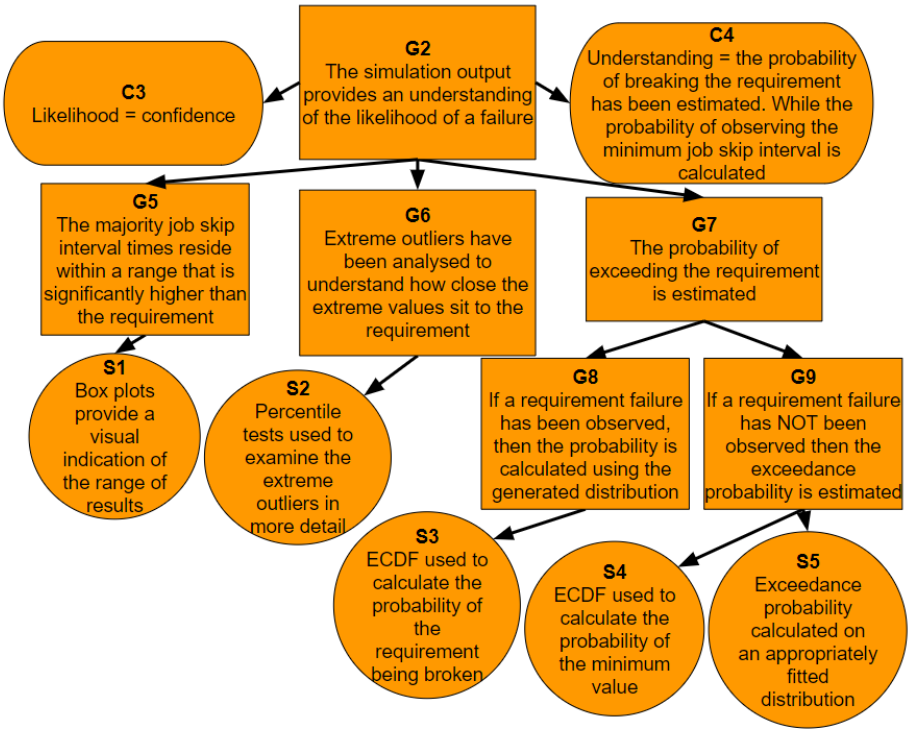


Figure 5.2: Goal Structured Notation Argument Exploring the Probability Assessment of the Requirement.

Ultimately, the aim of this assessment is to identify whether periods of time when the low DAL tasks are disabled are acceptably infrequent according to the Specification.

***How frequently will we fail to comply with the requirement?***

In line with Goal G7; once an evaluation has been performed to provide confidence that breaches of the requirement are rare, the next step is to attempt to quantify the actual probability of breaking the requirement.

Understanding the probability of breaking the requirement is assessed in one of two ways. If the requirement has been broken during testing, then the probability of this exceedance is estimated using an Empirical

Cumulative Distribution Function (ECDF), in line with Goal G8. However, if the requirement has not been broken then the exceedance probability is estimated using an extreme value theory on a fitted distribution of job skips, in line with Goal G9.

Goal G8 assumes that enough information (i.e. enough requirement compliance failures) has been provided by the simulation to fit a distribution and to read a result directly from the fitted distribution. If the simulation does not provide enough information to perform this assessment (i.e. there are insufficient failures to provide confidence in a directly read result), then G9 performs a statistical inference using the fitted distribution that aims to assess the tail of the distribution to understand the potential exceedance. This is read by assessing the job skip frequency.

### 5.1.4 Confidence

An evaluation of the results, as discussed in the previous section, can only be trusted if we can have confidence that the statistical analysis is performed across a significantly large sample that represents the real performance of the system. Goal G3 seeks to confirm this is the case and aims to understand whether enough testing has taken place.

Figure 5.3 shows the extension to Goal G3. This goal is fulfilled by ensuring the simulation executes for long enough to indicate that most execution time variations have been observed (G10), and that further exploration of the search space does not reveal new results (G11).

***Has the simulation executed for long enough?***

Goal G10 is concerned with understanding whether a single simulation executes for long enough, and is supported by an assessment that reviews whether continued simulation reveals any additional differences

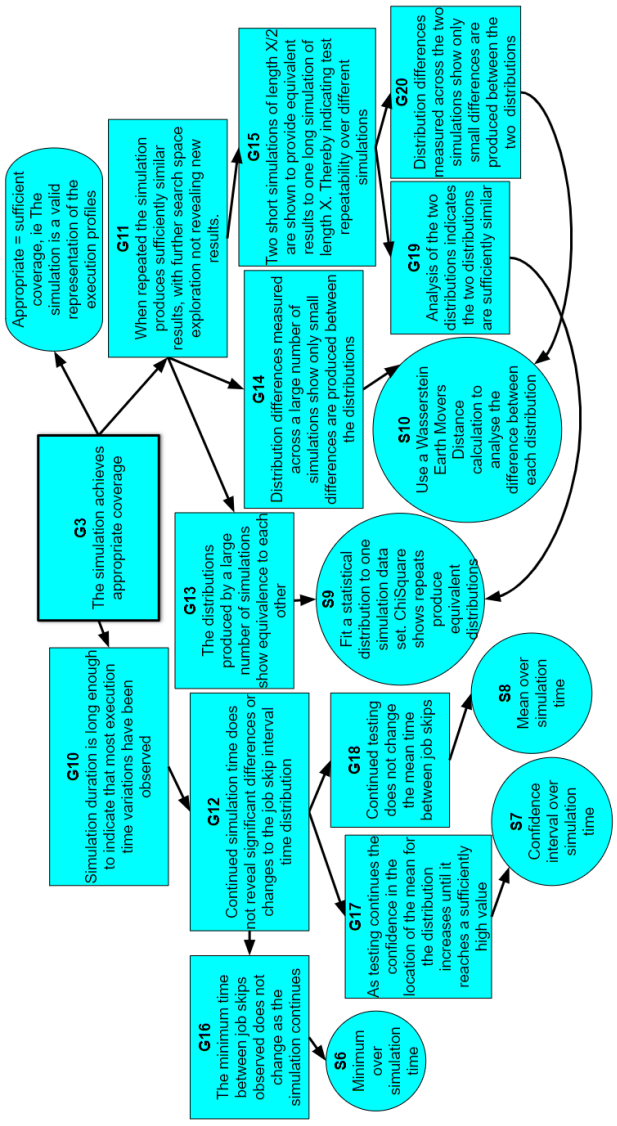


Figure 5.3: Goal Structured Notation Argument Exploring the Confidence of the Analysis.

or significant differences in the distribution. This is important to understand as it helps build the argument that the statistical analysis is performed across a fully representative set of execution profiles. This is tested by reviewing the minimum time between job skips (G16, S6), as well as the confidence interval (G17, S7) and the mean (G18, S8). In all

cases the aim of the assessment is to review whether, as the simulation continues, the results have converged.

***Does a large scale evaluation reveal different results?***

Goal G11 is concerned with understanding whether a large scale evaluation over a large number of simulations produces a similar result to that of a single simulation. This aims to provide further confidence that the search space has been explored sufficiently. The goal confirms firstly whether the analysis is repeatable when a large scale simulation is performed, and secondly that the results from multiple short simulations create a combined result equivalent to one long simulation.

Goal G11 is supported by an equivalence test of the job skip distributions over 100 (independently seeded) executions of the simulator using both a $\chi^2$ distribution test (G13) and an Earth Movers Distribution (EMD) test (G14). In both cases the simulation from the first test is used for comparison against the other 99. Secondly, G15 claims that when two short simulations are appended together they provide equivalent results to one long simulation. This analysis also uses the $\chi^2$ and EMD equivalence tests.

The $\chi^2$ distribution equivalence test provides an assessment of whether the two simulation distributions have been formed from the same master distribution. That is, are the two distributions independent, or equivalent to one another. The use in this context allows the assessment to confirm that a repeat of the simulation yields the same or equivalent results. The EMD test is so called based on an analogy of how it operates. The question posed by the test is given two mounds of earth, or soil, how much soil needs to be moved from one mound to the other before the two mounds are equivalent. Used alongside the $\chi^2$ test, the statistic provides further confidence that a repeat of the simulation produces equivalent results.

### 5.1.5 Validation

The process so far has focused on a simulation of the system. This is advantageous as the simulation can provide a much larger data set to analyse than is possible from execution on a real system test rig. Secondly, the results can be generated much faster than possible on real hardware. However, it is important to review the results, and the simulator, to ensure they reflect a valid representation of the real system.

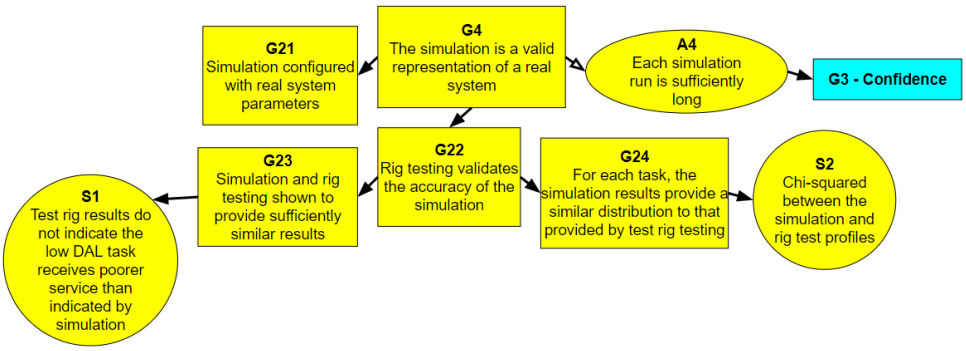


Figure 5.4: Goal Structured Notation Argument Exploring the Correctness of the Analysis.

Figure 5.4 extends the GSN argument and examines how the analysis provides representative results of the actual system performance. The goal has two steps. Firstly, it confirms that the initial simulation is configured with representative timing profiles. Secondly, the goal is verified using real results obtained from test rig operation (G22).

***Is the simulation configured correctly?***

Goal G21 concerns the input timing profiles used to generate the simulator results. As noted in the introduction to this section, the initial simulation should be set up using a set of task timing profiles generated through task-level execution in a representative environment, as detailed

further in Chapter 3. These timing profiles provide a representative set of results for the scheduler simulation to randomly iterate over.

Once a full system test rig campaign has been completed, the results from the real system should be used to both improve the simulator and to compliment the simulation produced results in order to improve accuracy. This full system test campaign is expected to provide a significantly larger set of results to boost confidence in the statistical analysis, arguably approaching a point where the simulation may not be required. However, these results would be expected to take significantly longer to generate, and would be provided at a time in the software design life-cycle too late to allow for cost effective improvement.

One risk with this approach is that the test rig campaign may indicate the simulation is flawed. This is a significant risk with any approach utilising a simulator and is in this case unavoidable. Nevertheless, the risks are mitigated by the fact that, as is frequently the case, the software project contains a number of legacy components for whom timing data should exist. The risk is further mitigated by an assumption that the simulation can be refined as soon as software testing begins rather than waiting for its completion. The key is that the simulation provides an easy environment for fast and efficient whole system (repeated) analysis.

***Does real world execution match the simulation?***

The second step to understanding if the results represent the real system is to compare a set of the produced simulation results against results obtained from the real system to ensure that they are both sufficiently similar. To do this, a subset of test rig results should be used to repeat the distribution analysis conducted to confirm Goal G11 in Figure 5.3. This is in order to verify that the sub-set of test rig results produce a similar distribution to the super-set of simulation results.

### 5.1.6 Summary

This section has presented a process for assessing the service provided to a low DAL task within an MCS. The process utilises a scheduler simulator, seeded with real system parameters, to perform an extensive statistical analysis of potential system execution profiles. The next section now applies the process to an industrial case study to examine how the process performs against both the AMC+ and resilient scheduling models.

## 5.2 Industrial System Use Case Application

This section now presents the results from applying the process for assessing low DAL service introduced in the previous section to a Rolls-Royce use case taken from the existing system introduced in Section 2.

### 5.2.1 Simulator Configuration

To facilitate this assessment the following process was followed for defining execution profiles:

- The Rolls-Royce aircraft engine control system task set introduced in Chapter 2, and ported to a mixed criticality system in Chapter 4, was imported into the scheduler simulator introduced in this chapter. Both the AMC+ and resilient models were implemented.

- The RTOS overheads for each scheduler implemented were measured using the process defined in Chapter 4. The overheads were measured as the system executed on the Rolls-Royce in-house processor.

- The system test High Water Marks (HWM) and analysed WCETs were used for $C_{LO}$ and $C_{HI}$ respectively.

- The timing profiles provided from Chapter 3 were used to generate a 'rate of timing fault' as follows: for the BCHLr fitness function the 95% measured execution time was treated as $C_{LO}$, with the number of times the fitness function obtained a time greater than 95% of the maximum measured execution time treated as a fault. This counter was then used to produce the 'rate of timing fault' for each task.

- On each release of a task the 'rate of timing fault' was used by the pseudo-random simulator to choose the job's execution time. If no fault was selected for this invocation, then a random number between the tasks Best Case Execution Time (BCET) and $C_{LO}$ was chosen. Otherwise, if a fault was selected a random number between the task's $C_{LO}$ and $C_{HI}$ was used.

- In order to ensure these execution profiles did not adversely skew the obtained results, or process; a high 'rate of timing fault' profile and low 'rate of timing fault' profile were also created. These were calculated as 10*[rate of fault] and 0.1*[rate of fault] respectively.

The simulator was executed on a high performance server in order to build up a comprehensive set of results. The execution time of each task was output by the simulator, as was information on whether a task executes, or is blocked. This data was analysed to measure the time between low criticality, or robust task job skips. A single execution simulates thirty minutes of scheduler time.

This section now progresses to examine the use case in detail. The **Specification** for the use case is defined, before an **Evaluation** of its

low DAL task service and an argument surrounding **Confidence** in the simulation is built. At this time as system level test results and timing profiles are not available, **Validation** of the digital twinning simulation approach is left to future work.

### 5.2.2 Non-Volatile Memory Access

In order to provide a secure record of engine performance, the control system regularly writes system parameters to flash memory. While non-volatile and secure, the time taken to write to this flash memory is considerable, with the task's execution time being directly proportional to the amount of data being written. Therefore, the amount of data written to the data store is minimised as far as possible; essentially its WCET is restricted. However, in order to support future design and maintenance goals, it is desirable to reduce this limitation.

The control system contains a periodic task responsible for writing data to flash memory. This task reads from a memory buffer written to by other tasks, before copying the buffer to the flash memory. At present the task is developed as a high DAL component and treated as a hard real time task. However, the task could more easily be designed to execute for longer, with an assumption that it may periodically drop jobs. Secondly, as the flash memory records are not used during flight, but instead for maintenance, the task could be developed against lower criticality processes. This assumes the necessary protection mechanisms are put in place, as defined in Chapter 4, to protect the wider system.

The flash memory task has been ported into the Rolls-Royce system discussed in Chapter 4, where it is treated as a robust low DAL task. The schedulability analysis for the control system was updated, while the period of the flash memory task was decreased, and the execution

time increased. Overall this increased the permissible utilisation of the task by a factor of 60. This increase was only permitted thanks to the MCS's exploitation of the difference between the analysed (sound, safe and pessimistic) WCET used for the $C^{HI}$ and the (test measured, robust but potentially optimistic) system test measured high water mark time used for each task's $C^{LO}$.

The system was implemented using both the AMC+ and the resilient models. In both cases the task set was clustered using the Deadline_P clustering technique. The AMC modelled system was shown to be schedulable in the low DAL mode, high DAL mode and during a mode change from the low to high modes. The resilient model was shown to be schedulable in the low DAL, fail robust (F-mode), fail resilient (M-mode) and high DAL modes; as well as the transitions between each mode, as defined by the robust model [34].

The following subsections now explore the service provided to the Non-Volatile Memory (NVM) task, following the process defined in Section 5.1.

#### 5.2.2.1 Specification

The newly configured flash memory task is designed to continuously write data when called to do so. If the task misses an execution then it will simply resume writing to memory from the next entry in the memory buffer. The principal requirement is that the memory buffer does not overflow; and so the task is designed to write more data than necessary on each invocation. This means that following a period of reduced service the task is able to progress back to normal operation provided it has time to recover.

The following assumptions surrounding the task have been defined

for this analysis:

- Due to the task's increased execution time, if given full service the task is capable of writing data to flash memory at a faster rate than the reporting tasks can write data to the shared memory buffer.
- The shared memory buffer is sufficiently large to allow the flash memory task to skip up to four jobs.
- Once the flash memory task skips a burst of up to four jobs, the task must execute the following four jobs for at least $C^{LO}$, in order to ensure no data is lost.
- Data loss is highly undesirable, but does not affect the safety of the system.

Therefore, the overriding requirement for analysis is that each time the flash memory task suffers a job skip burst, it should have a clear period of at least four successful executions before it can skip a job again. If the task skips a job in less time, the task is said to have suffered an error. The task period itself is 12.5ms; therefore the basic requirements for the task can be defined as follows:

**Definition 5.1.** A flash memory task error is recorded when the task suffers two separate bursts of job skips within 50ms.

**Definition 5.2.** A flash memory task error is recorded when the task suffers more than four consecutive job skips within a job skip burst.

The NVM task is treated as a robust low criticality task. So when executed using the AMC model, the task will be instantly disabled when the system moves to the high DAL mode. In the resilient model the task will skip up to four jobs when in the Resilient Mode. The task is then

disabled fully when in the high DAL mode. The static schedulability for the system, using both models, was confirmed using the processes discussed in Chapter 4.

#### 5.2.2.2 Evaluation

The case study for the Non-Volatile memory access use case was configured inside the scheduler simulator and tested against both the AMC+ and resilient models. The analysis was applied three times using the low, medium and high fault rates as introduced in Section 5.1.

***Are we confident the requirement will be complied with?***

Figure 5.5 shows the range of results obtained during one simulation of the NVM Case Study executing inside the AMC model with the high fault rate timing profile set. The main aim of reviewing the figure is to assess how far from the minimum requirement the majority of the inter-quartile range lies. In particular, to provide confidence; the majority of results should lie well above the requirement.

To further understand the extreme values in the simulation a percentile test is then applied to the full set of 100 simulation results obtained in Section 5.3. The results provide an assessment of the extreme minimum values obtained during simulation, as well as a measure of how close to the minimum requirement the majority of results lie. For example, the 0.1% percentile indicates how many results lie in the bottom 0.1% of the simulation results, showing a result expected at a frequency of 1 in every 1000 results.

The percentile test results for the NVM case study are shown in Table 5.1. The results indicate that, for all task timing profiles, the AMC scheduling method produced systems that would be expected to fail to comply with its NVM temporal requirements at a rate of 1 in every 200

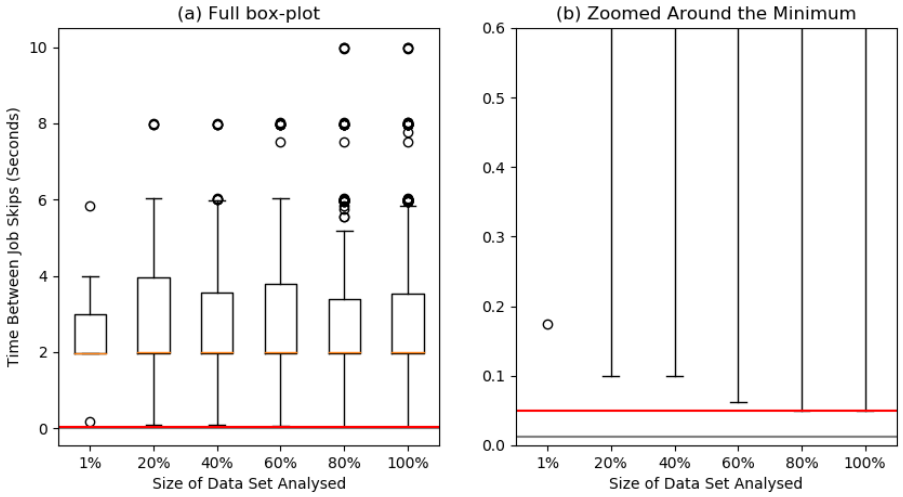


Figure 5.5: Box Plot Diagrams Showing the Range of Job Skip Interval Times, With a Zoomed-Plot on the Right Around the Minimum Requirement (AMC High Failure Rate).

times (based on the 0.5% outlier). The resilient scheduling model results however indicate that with the high error profile, the minimum time between job skips was measured as 150 seconds, well within the 50ms requirement. For the medium and low error profiles executing within the resilient scheduling model, no task job skips were observed.

***How frequently will we fail to comply with the requirement?***

To assess Goal G8 and Goal G9, the results of one simulation (for each scheduler mode and timing profile) was fitted to an exponential distribution in order to produce a continuous distribution for analysis. Figure 5.6 shows a histogram summarising results randomly selected from this fitted distribution, against a set of results randomly selected from the actual distribution. This figure shows a profile taken from the AMC high fault rate. This fitted distribution when compared to the original distribution

| % | Time Between Job Skips | | | | | |
|---|---|---|---|---|---|---|
| | AMC | | | Resilient | | |
| | High | Med | Low | High | Med | Low |
| 0.1% | 10.2ms | 10.9ms | 17.5ms | 150s | - | - |
| 0.5% | 10.6ms | 12.0ms | 33.6ms | 154s | - | - |
| 2.5% | 11.6ms | 24.5ms | 124ms | 168s | - | - |

Table 5.1: Percentile Outlier Assessment for the NVM Case Study.

provided a significance result of $\chi^2(12, n = 275) = 35.7, p < 0.01$[1]. It therefore indicated that the fitted distribution and actual distribution are both taken from the same population.

Reviewing the other simulation models and timing profiles; the results for each simulation were fitted to an exponential distribution, which was used to assess the probability of a requirement failure. These results are shown in Table 5.2. Secondly, Table 5.3 indicates the rate of failure for each distribution. This is based on the number of job skips observed during $10^9$ second timeframe, as obtained from a fitted distribution of job skip intervals.

A number of observations can be made from these results. Firstly, reviewing the analysis technique, the resilient model does not provide a high job skip rate. This in turn means the fitted distributions are applied using less data, reducing their validity. Conversely though, the fact the resilient models observe less failures provides a positive indication that the initial design provides sufficient service for the NVM task. At this

[1] $\chi^2$ results throughout this thesis are denoted using the following terminology - $\chi^2$([*degrees of freedom*], n = [*number of samples*)] = [*result*]), [*statistical significance*]. If the statistical significance (or p value) is less than 0.01, then the two compared distributions can be said to from the same population; that is they are not independent.

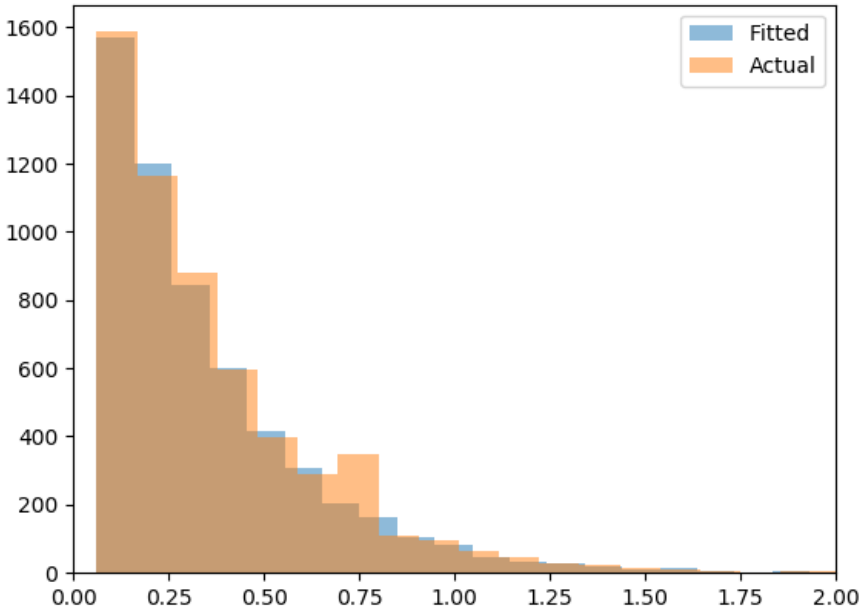


Figure 5.6: Histogram Illustrating the Difference in Results Randomly Selected From a Fitted Distribution and an Actual Distribution.

point in the design, as the simulation is based upon generated timing profiles, continuation of the simulation in an attempt to obtain further task failures would not necessarily offer further benefit. This question is explored further in the next section as **Confidence** surrounding the simulation search space is assessed. As the **Validity** of the approach is explored, and the generated profiles are combined with actual profiles obtained from a system test campaign, it is expected that the simulation would be repeated and extended to aim to provide further confidence in the design.

While the results generated should be considered in the context that they are based on manually generated timing profiles; it can be observed

| Skip Interval | % Results AMC High | AMC Med | AMC Low | Resilient High | Resilient Med | Resilient Low |
|---|---|---|---|---|---|---|
| 50ms | 53.4% | 7.5% | 0.69% | 0% | 0% | 0% |
| 60ms | 61.4% | 9.3% | 0.87% | 0% | 0% | 0% |
| 70ms | 68.1% | 11% | 1.04% | 0% | 0% | 0% |
| 80ms | 73.5% | 12.7% | 1.2% | 0% | 0% | 0% |
| 90ms | 78.1% | 14.4% | 1.4% | 0% | 0% | 0% |
| 100ms | 81.9% | 16% | 1.6% | 0% | 0% | 0% |
| 1s | 100% | 85.3% | 16% | 0% | 0% | 0% |
| 2s | 100% | 97.9% | 29.7% | 0% | 0% | 0% |

Table 5.2: Exceedance Probability from a Fitted Distribution of Simulation Results for the NVM Case Study.

| | High | Med | Low |
|---|---|---|---|
| AMC | 15981 | 1901 | 175 |
| Resilient | 1.4 | 0 | 0 |

Table 5.3: Failure Rate Assessed from Extended Simulation. Number of Failures per $10^9$s for the NVM Case Study.

that the use of graceful degradation in the resilient model has a significant effect on the rate of requirement failure. During simulation it was observed that while it is frequently the case that one task may overrun, it is rarely the case that multiple tasks overrun. This means that the AMC model, that switches to the high criticality mode after a single task has overrun, is severely disadvantaged.

#### 5.2.2.3 Confidence

Now an evaluation of the simulation results has been performed, the next step is to confirm whether the simulation has performed a valid search of the possible result space - in essence; will further testing reveal additional results?

***Has the simulation executed for long enough?***

Confidence in the results is assessed by confirming convergence, and secondly by reviewing multiple executions of each simulation to ensure the results showed equivalence. Tables 5.4 and 5.5 show the Minimum and Mean time observed during this assessment, and following convergence, between job skips in each setup. In all cases the results converge around these values, as is illustrated by Figures 5.7 and 5.8, which shows the results from one execution of the simulator for the AMC high error profile. The figure illustrates the variation in the confidence interval, mean and minimum as the simulation progresses. The results show that despite a significant amount of variability initially, the confidence interval (the range within which there is 95% confidence that the mean resides within) converges to less than 1ms. The mean and minimum converge to 62.4ms and 9.5ms[2] respectively. The key to analysing these plots is to identify whether the simulation results are changing as the simulation continues, or in essence do the results indicate that further exploration does not reveal any new or different results.

Tables 5.4 and 5.5 show how the decreasing task fault rate affects the AMC protocol, with the mean time between errors increasing signifi-

[2]This means the minimum time observed between job skips was actually less than the task's period. This was found to be due to task release and completion jitter thanks to the variation in execution time of higher priority tasks, some of which have longer periods.

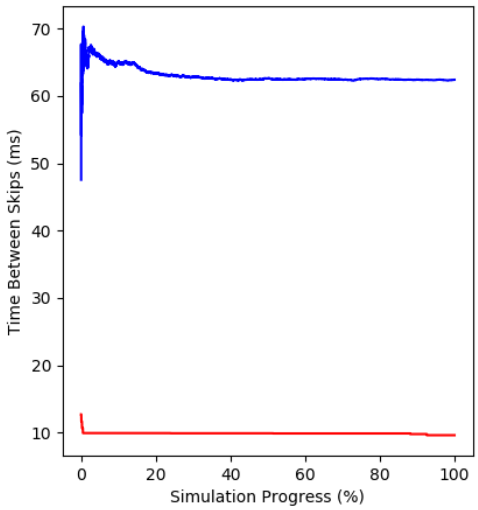


Figure 5.7: Changes in Mean (top) and Minimum (bottom) of the Time Between Job Skip Bursts Over Simulation Time.

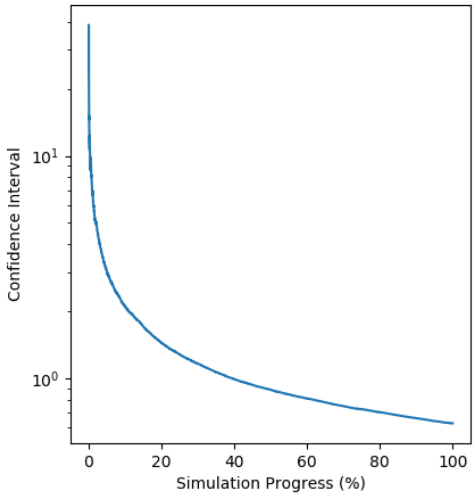


Figure 5.8: Changes in Confidence Interval of the Time Between Job Skip Bursts Over Simulation Time.

| | High | Med | Low |
|---|---|---|---|
| AMC | 9.6ms | 9.8ms | 14.2ms |
| Resilient | 32,012ms | X | X |

Table 5.4: Minimum Time Between Requirement Errors For The Non-Volatile Memory Access Case Study.

| | High | Med | Low |
|---|---|---|---|
| AMC | 62.4ms | 504.9ms | 5,328.2ms |
| Resilient | 250,392ms | X | X |

Table 5.5: Mean Time Between Requirement Errors For The Non-Volatile Memory Access Case Study.

cantly, reducing the probability of breaches of compliance of the requirement. However, even with a low error rate the AMC model still produces a system where a time between job skip bursts of approximately 14.2ms can be observed.

The resilient model however, when using the same task timing profiles, produces a system where the time between job skips is significantly higher. In the case of the Medium and Low error profiles, the simulator did not observe a requirement error; while the resilient model with a high task fault rate observed Minimum and Mean times between requirement errors of approximately 4000 times lower than the same failure rate observed with the AMC model.

Reviewing the technique used for assessing low DAL task service; in order to identify whether the simulation of the resilient model ever observes a requirement error, the simulation was repeated for a longer period of time (equating to approximately 900 minutes of processor time). Again the simulator did not observe a job skip at all during this time.

Indeed, it was confirmed that for both the medium and low timing profiles executing within the resilient model that the model did not enter the resilient mode, let alone the high DAL mode.

***Does a large scale evaluation reveal different results?***

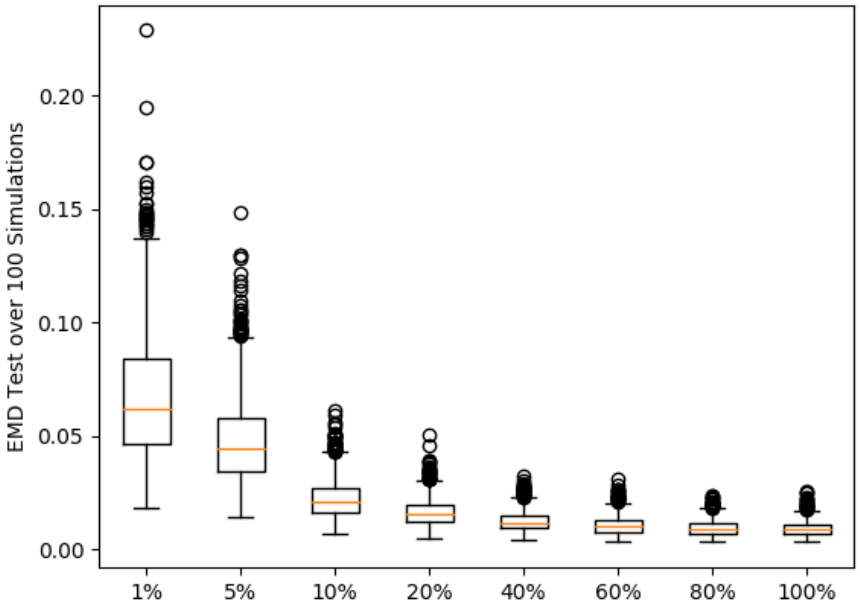


Figure 5.9: Comparison of EMD over 1000 Simulations.

Figure 5.9 shows the EMD result from executing 1000 simulations. In each case each simulation's distribution was randomly sampled using different sample sizes of the set (1%, 5%, 10%, 20%, 40%, 60%, 80%, 100%) of the length of the distribution. This randomly sampled set was then compared, using an EMD test, to a randomly selected distribution of the same length taken from the first simulation. As can be seen from Figure 5.9 the larger the chosen sample, the closer the two randomly selected distributions. Secondly, the results are shown to converge as more data is appended to the sample.

Furthermore, the distribution of the first simulation was fitted to an exponential distribution in order to produce an expected distribution to test against (fitted with $\chi^2(12, n = 275) = 34, p < 0.01$). Each of the other 999 distributions produced by the simulations were then compared to this fitted distribution using a $\chi^2$ distribution equivalence test; which showed each simulation was produced from the same population (mean result - $\chi^2(12, n = 275) = 32.5, p < 0.01$).

#### 5.2.2.4 Process Review

These results have indicated that the NVM task can be implemented inside the Rolls-Royce system profiled in this thesis. The analysis conducted allowed the task's permissible execution time to be expanded by a factor of 60, with the timing profiles and simulation providing an initial indication of the failure rate to be expected for the task.

Reviewing the approach in general; one risk is that some of the statistical methods used (for example extreme value theory) assume that the input statistical profile is independent and identically distributed. It is necessary to perform further work to assess that this is indeed the case with the generated timing profiles. In the meantime to reduce the risk of this approach the preferred option would be to infer results directly from the generated simulation profile. Following on from Goal G7 (Figure 5.10) the preferred solutions would be S3 and S4, rather than solution S5.

The next steps in the process for assessing this case study would be to update the simulation based on results obtained from a real system test campaign, as well as assessing the service received by the task directly inside the integrated system. Whether the task receives appropriate service would then have to be assessed from a safety, and mission critical

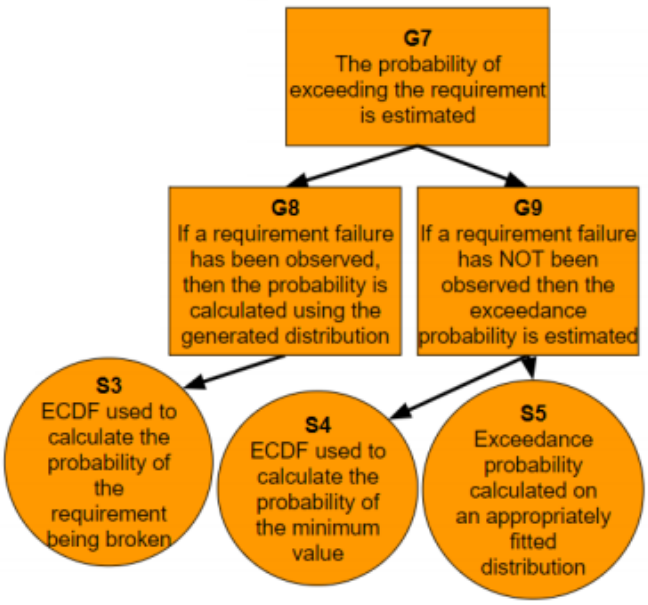


Figure 5.10: Assessing the Probability of Failure.

point of view. This would require the probability of failure (the risk) being played off against the consequence of failure.

## 5.3 Summary

This chapter has defined a process that may be applied to assess the service afforded to a low criticality task in an MCS. The process uses a scheduler simulator to feed a statistical analysis which aims to provide a rate of failure for each low criticality task. This result can then be used to assess compliance to requirements, and ultimately provide a level of confidence that the task's temporal requirements will be met. Crucially, the process is designed to be performed early in a design lifecycle to allow for fast and efficient redesign if necessary. As project maturity builds, the statistical analysis can be seeded with execution time distributions obtained from system test results to refine and build confidence in the

produced results.

The process has been applied to the system developed in Chapter 4, and is designed to use the results output from the analysis introduced in Chapter 3. A use case, provided by the real industrial system discussed in Chapter 2, was analysed against the approach to attempt to assess its applicability. The analysis, applied across two different scheduling models with three different timing profiles, was able to provide a failure rate where task job skips has been observed. For the systems that exhibited lower task job skip rates, the statistical analysis process provided less indicative results. However, this in itself is a positive result for the resilient scheduling system tested.

The next steps for this analysis would be to extend the process with results obtained from a comprehensive system test programme, the aim being to build confidence in the failure rates observed.

# Chapter 6

# Conclusions and Future Work

The motivation for this work comes from a desire to reduce software cost through the introduction of improved tools and processes to increase software development efficiency. The key focus of this thesis centred on the development and proof of mixed criticality systems to achieve this aim.

Mixed criticality systems and mixed criticality scheduling techniques offer the potential to better utilise processor hardware by capitalising on WCET pessimism; and allow the use of cost effective appropriately levelled software development processes. However, a number of open problems exist which block the adoption of MCS technology. These include how to effectively analyse the timing performance of tasks within such a system, how to develop and certify systems and how to prove the service provided to low DAL tasks. This thesis has aimed to address these issues. Crucially, the work has been tested on a real full scale industrial system without simplification. The following sections now discuss the work completed within this thesis.

## 6.1 Review of Work Completed

Chapter 1 reviewed the issues currently faced by software developers and set the scene for the improvements assessed in this thesis. In particular it outlined that safety critical software applications are expensive to develop, in some cases to such a point that innovation and advancement are impacted. Current development methods rely on systems developed to the same criticality. This increases development costs as some software components are developed against stricter standards than necessary. Furthermore, timing analysis processes can be expensive to perform, while providing pessimistic results too late in the design lifecycle. These pessimistic results, while often necessary for safety, are not appropriately accommodated by current scheduling techniques.

Chapter 2 introduced an example industrial system, used throughout the following chapters for testing the developed research. This system was taken from a live industrial project. The application of the real time system research developed throughout this thesis to this real industrial example forms one of the contributions of this work. Which represents, to the best of the author's knowledge, one of the first examples discussing the application of this technology to a system of this scale.

Chapter 3 focused on the assessment of the timing properties of tasks within a system. Reviewing the existing literature reveals that current processes and practices for measurement, or hybrid based WCET analysis, assume that test data to support analysis is already present. Whereas existing methods of automatic test case generation, to a certain extent, assume that the WCET will be stumbled upon. The work in this thesis suggested that the combination of the two processes would produce a sound method for analysis.

This led to one of the core contributions of the chapter; the develop-

ment of a series of fitness functions based not on previous techniques for automatic software execution, but on an understanding of the requirements for hybrid measurement based WCET analysis. The new fitness functions focus on confidence (in the form of coverage), and not on execution times observed. The algorithms were tested against a series of industrial examples, as well as a set of academic benchmarks, and were shown to provide superior results to the previously accepted methods.

The key output of the tooling in this section is a method for generating timing properties and profiles for each task within a system, that allows the system proof in later chapters to be performed.

Chapter 4 progresses to review MCS development. A review of the current work in the field of MCSs reveals that while much work has been performed on the development of scheduling methodologies, there has been less work on the side of system configuration to support certification. The chapter reviews the certification requirements for a MCS, before presenting the design for an architecture aiming to provide the temporal and spatial partitioning required to support timing efficient development of a MCS for a high/low criticality avionics application. This, together with appropriate processes for handling scheduler overheads, forms one of the contributions of this chapter.

The chapter then examines how an existing system may be ported into the new MCS architecture. The process reviews the temporal requirements of the system in question, which includes a complex task set interlaced with jitter, transactional and periodic requirements. This led to the development of a legacy system porting process. The porting process, which aims to define a schedulable system with minimal overheads, takes each requirement into account automatically to help define the final system. This forms the second contribution of the chapter. The process

defined is assessed against two industrial examples before being applied to a large scale randomised assessment.

Finally, Chapter 5 brings together the WCET assessment work in Chapter 3 with the MCS design effort in Chapter 4 to present an approach for assessing the service provided to low criticality tasks. The need for the work is identified from an assessment of the available literature, which found significant gaps around how to gain an understanding of the performance of low criticality tasks within an MCS. The approach, which forms the principal contribution of the chapter, relies on a statistical assessment of the results produced by a scheduler simulator. The simulator was fed with the analysis results from Chapter 3 and the scheduler configuration information obtained from Chapter 4. Ultimately, the process defined is discussed in the context of a real industrial case study taken from the system introduced in Chapter 2.

## 6.2 Future Work

The work discussed in this thesis has been targeted at application on an industrial project, and has been tested as such. However, there remains future work required before such a system could be used for certification on a live project.

The analysis in Chapter 3 has at time of writing been advanced the furthest towards application on an industrial project, and has provided timing results to support certification. However, at present this work has focused on deterministic architectures, using the processor introduced in Chapter 2. Lesage et al. looked at whether the fitness functions developed in this thesis could be scaled to a whole system [67]. Lesage found that the approach was scaleable, however future work on this subject

would benefit from looking at more advanced architectures, or fitness function combinations. In particular whether any advanced processor features, such as caches, could be used to better target the generation of measurement based timing analysis profiles.

This requirement to review the approaches applicability to more advanced architectures can also be extended to the work discussed in Chapters 4 and 5. Focusing on a real industrial processor has allowed an in-depth analysis of a real system to be conducted. However, this approach does carry the risk that the work presented in this thesis is targeted, or focused, on this one processing platform. While this is a real industrial platform of the type used across industry, it is still important that future work focuses on identifying whether any assumptions made through this thesis need to be reexamined.

The system introduced in Chapter 4 has targeted development using fixed priority schedulers. Such schedulers were chosen based on the requirement for ease of verification and understanding. The chapter hypothesises that the clustering and partitioning designs introduced would be applicable to dynamic priority systems, such as those utilising earliest deadline first schedulers. However, no work has been undertaken to support this hypothesis; it is at present left to future work.

Additionally, the assessment of RTOS overheads assumes an easy to verify RTOS, executing upon a deterministic processor. The first assumption is perhaps valid given safety critical applications. However, as software requirements grow, assessment of RTOS overheads on less deterministic processors may need to be addressed. The work in chapter 4 is clear in its requirement that the overheads are either assessed or bounded. So purely bounded overheads may help this future requirement; however it could also be a source of pessimism in the approach.

Chapter 5 presented an approach for validating low critically task service using a system scheduler. One key point of future work identified by the chapter is that the process of validating the scheduler simulator itself is currently left to future work. The process requires a full set of results produced by the work in Chapter 3 and the full set of results produced by an end to end software test campaign. At time of writing such results are not available for assessment.

Finally, the statistical assessment presented in Chapter 5 assumes that task job failures, or overruns, are independent and identically distributed. In the system explored in this thesis this can be said to be the case. However, for systems where this may not be the case the statistical assessment may need to be expanded. This is at present left to future work.

## 6.3 Final Remarks

The central proposition of this thesis is:

*Automatic test case generation techniques can be extended to reliably target hybrid measurement based timing analysis to produce sound WCET profiles. These produced WCET profiles can then be used to aid the development and validation of mixed criticality schedulers, provided the certification objectives, overheads of the scheduler, and the service provided to low criticality tasks are not neglected.*

This thesis has shown that appropriately targeted search algorithms can be used to guide measurement based WCET analysis. This assertion is backed up by the analysis conducted in this thesis, but also by the work conducted by Lesage [67] which used the approach to analyse a full aircraft engine control system on a parallel project to the system used in

this thesis.

A Mixed Criticality System design has been presented, developed to comply with certification objectives to support full start to end system development. An existing system has been ported to this new architecture (without simplification or modification).

This scheduler design has since been ported into the new Rolls-Royce software architecture. At this time, the scheduler will not be used to allow development and integration of a mixed criticality system. However, it will be used to control debug and test functionality to ensure their operation does not impact the system they are analysing. This will allow the approach to be further reviewed and refined in practise.

Finally, a process for reviewing the service afforded to low criticality tasks has been presented. The process utilises the WCET profiles obtained from Chapter 3 and provides a measure on the probability of requirement failure for a set of low criticality components within the developed MCS. The approach is developed around a goal structured safety case, constructed and supported using a series of statistical analyses.

Additionally, while not a key output from this thesis (as the timing profiles used to support the analysis were derived), the results of applying the approach discussed in this thesis to a real industrial system allowed a substantial increase in available processor utilisation. The results indicated that a MCS utilising a robust system architecture was able to provide full service to a low criticality component, while allowing the introduction of an extra 40% system utilisation over the original single criticality system.

With these remarks in mind, it can be considered that the proposition of this thesis is valid.

# Appendix A

# Open Source Control System Example Taskset

This open source control system example taskset was taken from Bate [38].

| TaskID | Jitter | Period | Deadline | WCET |
|---|---|---|---|---|
| P1 | 0 | 25000 | 25000 | 300 |
| P2 | 0 | 25000 | 25000 | 2088 |
| P3 | 12500 | 25000 | 12961 | 461 |
| P4 | 0 | 25000 | 25000 | 340 |
| P5 | 0 | 25000 | 25000 | 7 |
| P6 | 0 | 25000 | 25000 | 85 |
| P7 | 0 | 25000 | 25000 | 1910 |
| P8 | 0 | 25000 | 25000 | 1971 |
| P9 | 0 | 25000 | 25000 | 640 |
| P10 | 0 | 25000 | 25000 | 17 |
| P11 | 12500 | 25000 | 13171 | 671 |
| P12 | 0 | 25000 | 25000 | 103 |
| P13 | 0 | 25000 | 25000 | 203 |

| TaskID | Jitter | Period | Deadline | WCET |
|---|---|---|---|---|
| P14 | 0 | 25000 | 25000 | 26 |
| P15 | 0 | 25000 | 12960 | 14 |
| P16 | 0 | 25000 | 25000 | 408 |
| P17 | 0 | 25000 | 25000 | 278 |
| P18 | 0 | 25000 | 25000 | 190 |
| P19 | 0 | 25000 | 25000 | 32 |
| P20 | 0 | 25000 | 25000 | 228 |
| P21 | 12500 | 25000 | 13184 | 684 |
| P22 | 0 | 25000 | 25000 | 273 |
| P23 | 0 | 25000 | 25000 | 1265 |
| P24 | 0 | 50000 | 12668 | 318 |
| P25 | 0 | 100000 | 12957 | 1334 |
| P26 | 0 | 50000 | 12669 | 52 |
| P27 | 0 | 200000 | 12958 | 796 |
| P28 | 0 | 50000 | 12958 | 336 |
| P29 | 0 | 50000 | 12958 | 408 |
| P30 | 0 | 50000 | 12670 | 798 |
| P31 | 0 | 100000 | 13182 | 457 |
| P32 | 0 | 50000 | 49999 | 351 |
| P33 | 0 | 50000 | 12671 | 390 |
| P34 | 0 | 50000 | 13181 | 201 |
| P35 | 12500 | 50000 | 12673 | 173 |
| P36 | 0 | 50000 | 50000 | 925 |
| P37 | 0 | 50000 | 50000 | 321 |
| P38 | 0 | 50000 | 12959 | 1801 |
| P39 | 0 | 50000 | 50000 | 522 |
| P40 | 0 | 50000 | 50000 | 256 |

| TaskID | Jitter | Period | Deadline | WCET |
|---|---|---|---|---|
| P41 | 0 | 100000 | 12960 | 196 |
| P42 | 0 | 50000 | 50000 | 900 |
| P43 | 0 | 50000 | 12959 | 1945 |
| P44 | 0 | 100000 | 13183 | 528 |
| P45 | 0 | 100000 | 12672 | 551 |
| P46 | 0 | 100000 | 100000 | 272 |
| P47 | 0 | 100000 | 100000 | 271 |
| P48 | 0 | 100000 | 100000 | 378 |
| P49 | 0 | 100000 | 100000 | 107 |
| P50 | 0 | 100000 | 100000 | 217 |
| P51 | 0 | 100000 | 100000 | 4698 |
| P52 | 0 | 100000 | 100000 | 232 |
| P53 | 0 | 100000 | 100000 | 30 |
| P54 | 0 | 100000 | 100000 | 763 |
| P55 | 0 | 100000 | 100000 | 62 |
| P56 | 0 | 200000 | 200000 | 304 |
| P57 | 0 | 200000 | 200000 | 336 |
| P58 | 0 | 200000 | 200000 | 100 |
| P59 | 0 | 200000 | 200000 | 8 |
| P60 | 0 | 200000 | 200000 | 378 |
| P61 | 0 | 200000 | 200000 | 38 |
| P62 | 0 | 200000 | 200000 | 428 |
| P63 | 0 | 200000 | 200000 | 2258 |
| P64 | 0 | 200000 | 200000 | 328 |
| P65 | 0 | 1000000 | 1000000 | 5040 |
| P66 | 0 | 1000000 | 1000000 | 5040 |
| P67 | 0 | 1000000 | 1000000 | 5040 |

| TaskID | Jitter | Period | Deadline | WCET |
|---|---|---|---|---|
| P68 | 0 | 1000000 | 1000000 | 5040 |
| P69 | 0 | 1000000 | 1000000 | 5040 |
| P70 | 0 | 1000000 | 1000000 | 5040 |
| P71 | 0 | 1000000 | 1000000 | 5040 |
| P72_low | 0 | 25000 | 25000 | 100 |
| P73_low | 15000 | 50000 | 15010 | 10 |
| P74_low | 0 | 50000 | 50000 | 3000 |
| P75_low | 0 | 100000 | 100000 | 5000 |

Table A.1: Example Control System Task Set.

# Bibliography


[1] R. F. Paige, A. Zolotas, D. S. Kolovos, J. A. McDermid, M. Bennett, S. Hutchesson, and A. Hawthorn, "SECT-AIR: Software engineering costs and timescales – aerospace initiative for reduction", *Software Technologies: Applications and Foundations*, M. Seidl and S. Zschaler, Eds., pp. 403–408, 2018.

[2] P. Feiler, J. Goodenough, A. Gurfinkel, C. Weinstock, and L. Wrage, "Four pillars for improving the quality of safety-critical software-reliant systems", Carnegie-Mellon University, Tech. Rep., 2013.

[3] D. Dvorak, "NASA study on flight software complexity", in *AIAA Infotech@Aerospace Conference*. 2009.

[4] P. Johnston and R. Harris, "The Boeing 737 MAX saga: lessons for software organizations", *Software Quality Professional*, vol. 21, no. 3, pp. 4–12, 2019.

[5] C Haddon-Cave, "The Nimrod Review–The Loss of RAF Nimrod XV230: A Failure of leadership, culture and priorities", *Report HC1025. London Stationary Office, Crown Copyright*, 2009.

[6] RTCA, "DO-178C - Software Considerations in Airborne Systems and Equipment Certification", 2011.

[7] SAE International, "ARP4754A - guidelines for development of civil aircraft and systems", vol. 12, 2010.

[8] B. W. Boehm, "Software engineering economics", *Prentice-hall Englewood Cliffs (NJ)*, vol. 197, 1981.

[9] P Herzlich, "The politics of testing", *Proceedings of the 1st European International Conference on Software Testing Analysis and Review*, 1993.

[10] R. Kasauli, E. Knauss, B. Kanagwa, A. Nilsson, and G. Calikli, "Safety-critical systems and agile development: A mapping study", *Proceedings of the 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 470–477, 2018.

[11] R. Kirner and P. Puschner, "Obstacles in worst-case execution time analysis", *Proceedings of the 11th International Symposium on Object Oriented Real-Time Distributed Computing*, pp. 333–339, 2008.

[12] S. Bünte, M. Zolda, and R. Kirner, "Let's get less optimistic in measurement based timing analysis", pp. 204–212, 2011.

[13] S. Law, M. Bennett, I. Ellis, S. Hutchesson, G. Bernat, A. Colin, and A. Coombes, "Effective worst-case execution time analysis of DO178C level A software", *Ada User Journal*, vol. 36, no. 3, pp. 182–186, 2015.

[14] N. Tracey, J. Clark, K. Mander, and J. McDermid, "An automated framework for structural test-data generation", *Proceedings of the 13th International Conference on Automated Software Engineering*, pp. 285–288, 1998.

[15] R. Davis, I. Bate, G. Bernat, I. Broster, A. Burns, A. Colin, S. Hutchesson, and N. Tracey, "Transferring real-time systems research into industrial practice: Four impact case studies", *Proceedings of the 30th Euromicro Conference on Real-Time Systems*, 2018.

[16] G. Bernat, R. Davis, N. Merriam, J. Tuffen, A. Gardner, M. Bennett, and D. Armstrong, "Identifying opportunities for worst-case execution time reduction in an avionics system", *Ada User Journal*, vol. 28, no. 3, pp. 189–195, 2007.

[17] J.-F. Deverge and I. Puaut, "Safe measurement-based WCET estimation", *Proceedings of the 5th International Workshop on WCET Analysis*, vol. 5, pp. 13–16, 2005.

[18] S. Stattelmann and F. Martin, "On the use of context information for precise measurement-based execution time estimation", *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, pp. 13–16, 2010.

[19] J. Wegener, H. Sthamer, B. F. Jones, and D. E. Eyres, "Testing real-time systems using genetic algorithms", *Software Quality Journal*, vol. 6, no. 2, pp. 127–135, 1997.

[20] B. Jones, H. Sthamer, X. Yang, and D. Eyres, "The automatic generation of software test data sets using adaptive search techniques", *Proceedings of the 3rd International Conference on Software Quality Management*, pp. 435–444, 1995.

[21] P. Graydon and I. Bate, "Safety assurance driven problem formulation for mixed-criticality scheduling", pp. 19–24, 2013.

[22] Aeronautical Radio Incorporated, "Avionics application software standard interface part 1 - required services", *ARINC Specification 653 Part 1-3, Aeronautical Radio, Inc.*, 2010.

[23] N. Audsley and A. Wellings, "Analysing APEX applications", *17th IEEE International Real-Time Systems Symposium, (RTSS)*, pp. 39–44, 1996.

[24] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance", *Proceedings of the 28th IEEE International Real-Time Systems Symposium, (RTSS)*, pp. 239–243, 2007.

[25] S. Baruah, A. Burns, and R. Davis, "Response-time analysis for mixed criticality systems", *Proceedings of the 32nd IEEE International Real-Time Systems Symposium, (RTSS)*, pp. 34–43, 2011.

[26] S. Baruah and A. Burns, "Implementing mixed criticality systems in ada", *Proceedings of the International Conference on Reliable Software Technologies*, pp. 174–188, 2011.

[27] I. Bate, A. Burns, and R. Davis, "A bailout protocol for mixed criticality systems", *Proceedings of the 27th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 259–268, 2015.

[28] M. Jan, L. Zaourar, and M. Pitel, "Maximizing the execution rate of low criticality tasks in mixed criticality system", *Proceedings of the 1st Workshop on Mixed Criticality Systems (WMC), RTSS*, pp. 43–48, 2013.

[29] G. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni, "Elastic scheduling for flexible workload management", *IEEE Transactions on Computers*, vol. 51, no. 3, pp. 289–302, Mar. 2002.

[30] D. Liu, J. Spasic, N. Guan, G. Chen, S. Liu, T. Stefanov, and W. Yi, "EDF-VD scheduling of mixed-criticality systems with degraded quality guarantees", *Proceedings of the 37th IEEE Real-Time Systems Symposium (RTSS)*, pp. 35–46, 2016.

[31] S. Altmeyer, B. Lisper, C. Maiza, J. Reineke, and C. Rochange, "WCET and mixed-criticality: What does confidence in WCET estimations depend upon?", *Proceedings of the 15th International*

*Workshop on Worst-Case Execution Time Analysis (WCET)*, pp. 65–74, 2015.

[32] H. Su and D. Zhu, "An elastic mixed-criticality task model and its scheduling algorithm", *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 147–152, 2013.

[33] C. Gill, J. Orr, and S. Harris, "Supporting graceful degradation through elasticity in mixed-criticality federated scheduling", *Proceedings of the 6th Workshop on Mixed Criticality Systems (WMC), RTSS*, pp. 19–24, 2018.

[34] A. Burns, R. Davis, S. Baruah, and I. J. Bate, "Robust mixed-criticality systems", *IEEE Transactions on Computers*, pp. 1478–1491, 2018.

[35] T. Kelly, "Arguing safety: A systematic approach to managing safety cases", PhD thesis, The University of York, 1999.

[36] S. Quinton, "Evaluation and comparison of real-time systems analysis methods and tools", *Proceedings of the International Workshop on Formal Methods for Industrial Critical Systems*, pp. 284–290, 2018.

[37] S. Law and I. Bate, "Achieving appropriate test coverage for reliable measurement-based timing analysis", *Proceedings of the 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 189–199, 2016.

[38] I. Bate, "Scheduling and timing analysis for safety critical real-time systems", PhD thesis, The University of York, 1999.

[39] I. Bate and A. Burns, "An integrated approach to scheduling in safety-critical embedded control systems", *Real-Time Systems Journal*, vol. 25, no. 1, pp. 5–37, Jul. 2003.

[40] B. Korel, "Automated software test data generation", *IEEE Transactions on software engineering*, vol. 16, no. 8, pp. 870–879, 1990.

[41] C Ballabriga, H Cassé, and M De Michiel, "The Mälardalen WCET benchmarks: Past, present and future", *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, pp. 136–146, 2010.

[42] S. Petters, "Bounding the execution time of real-time tasks on modern processors", *Proceedings the 7th International Conference on Real-Time Computing Systems and Applications*, pp. 498–502, 2000.

[43] *Rapitime explained: White paper*, Accessed: 2014-06-13. [Online]. Available: `http://www.rapitasystems.com/downloads/brochures-white-papers/rapitime-explained`.

[44] A. Betts, G. Bernat, R. Kirner, P. Puschner, and I. Wenzel, "WCET coverage for pipelines", *Real-Time Systems Research Group - University of York and Institute of Computer Engineering - Vienna University of Technology, Technical Report*, 2006.

[45] S. Edgar and A. Burns, "Statistical analysis of WCET for scheduling", *Proceedings of the 22nd Real-Time Systems Symposium (RTSS)*, pp. 215–224, 2001.

[46] J. Hansen, S. Hissam, and G. Moreno, "Statistical-based WCET estimation and validation", *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis*, pp. 1–11, 2009.

[47] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quinones, and F. J. Cazorla, "Measurement-based probabilistic timing analysis for multi-path programs", *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 91–101, 2012.

[48] S. J. Gil, I. Bate, G. Lima, L. Santinelli, A. Gogonel, and L. Cucu-Grosjean, "Open challenges for probabilistic measurement-based worst-case execution time", *IEEE Embedded Systems Letters*, vol. 9, no. 3, pp. 69–72, 2017.

[49] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, "A definition and classification of timing anomalies", *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis*, pp. 23–28, 2006.

[50] A. Colin and S. Petters, "Experimental evaluation of code properties for WCET analysis", *Proceedings of the 24th Real-Time Systems Symposium*, pp. 190–199, 2003.

[51] G. Bernat, A. Colin, and S. Petters, "PWCET: A tool for probabilistic worst-case execution time analysis of real-time systems", *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems*, 2003.

[52] S. Bünte, M. Zolda, M. Tautschnig, and R. Kirner, "Improving the confidence in measurement-based timing analysis", *Proceedings of the 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pp. 144–151, 2011.

[53] I. Bate and U. Khan, "WCET analysis of modern processors using multi-criteria optimisation", *Empirical Software Engineering*, vol. 16, no. 1, pp. 5–28, 2011.

[54] N. Williams, "WCET measurement using modified path testing", *Proceedings of the 5th International Workshop On Worst-Case Execution-Time (WCET) Analysis*, pp. 17–20, 2005.

[55] N. Williams and M. Roger, "Test generation strategies to measure worst-case execution time", *Proceedings of the ICSE Workshop on Automation of Software Test*, pp. 88–96, 2009.

[56] I. Wenzel, R. Kirner, B. Rieder, and P. Puschner, "Measurement-based timing analysis", *Leveraging Applications of Formal Methods, Verification and Validation*, pp. 430–444, 2009.

[57] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith, "FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement", pp. 209–213, Jul. 2008.

[58] R. Kirner and M. Zolda, "Compiler support for measurement-based timing analysis", *Proceedings of the 11th International Workshop on Worst-Case Execution Time Analysis*, pp. 62–71, 2011.

[59] M. Zolda and R. Kirner, "Calculating WCET estimates from timed traces", *Real-Time Systems*, vol. 52, no. 1, pp. 38–87, 2016.

[60] T. J. McCabe, "A complexity measure", *IEEE Transactions on Software Engineering*, no. 4, pp. 308–320, 1976.

[61] S. Kirkpatrick, D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing", *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[62] D. Connolly, "General purpose simulated annealing", *Journal of the Operational Research Society*, pp. 495–505, 1992.

[63] N. Tracey, "A search-based automated test-data generation framework for safety-critical software", PhD thesis, The University of York, 2000.

[64] RTCA, "DO-330 - Software tool Qualification Considerations", 2011.

[65] S Siegel and N. Castellan, *Non parametric statistics for the behavioral sciences*. McGraw-Hill International, 1988.

[66] M. Hollander and D. A. Wolfe, *Nonparametric statistical methods*. Wiley-Interscience, 1999.

[67] B. Lesage, S. Law, and I. Bate, "TACO: An industrial case study of test automation for coverage", *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, pp. 114–124, 2018.

[68] J. Rushby, "Partitioning for safety and security: Requirements, mechanisms, and assurance", NASA Langley Research Center, NASA Contractor Report, 1999.

[69] K. Tindell and A. Alonso, "A very simple protocol for mode changes in priority preemptive systems", Universidad Politecnica de Madrid, Tech. Rep., 1996.

[70] I. Bate, A. Burns, J. McDermid, and A. Vickers, "Towards a fixed priority scheduler for an aircraft application", *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, pp. 34–39, 1996.

[71] S. Baruah and S. Vestal, "Schedulability analysis of sporadic tasks with multiple criticality specifications", *Proceedings of the 20th Euromicro Conference on Real-Time Systems*, pp. 147–155, 2008.

[72] P. Ekberg and W. Yi, "Bounding and shaping the demand of generalized mixed-criticality sporadic task systems", *Real-time systems*, vol. 50, no. 1, pp. 48–86, 2014.

[73] N. Guan, P. Ekberg, M. Stigge, and W. Yi, "Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems", *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, pp. 13–23, 2011.

[74] P. Ekberg and W. Yi, "A note on some open problems in mixed-criticality scheduling", *Proceedings of the 6th International Real-Time Scheduling Open Problems Seminar*, pp. 1–2, 2015.

[75] P. B. Sousa, K. Bletsas, E. Tovar, P. Souto, and B. Åkesson, "Unified overhead-aware schedulability analysis for slot-based task-splitting", *Real-Time Systems*, vol. 50, no. 5-6, pp. 680–735, 2014.

[76] J. Freitag, S. Uhrig, and T. Ungerer, "Virtual timing isolation for mixed-criticality systems", *Proceedings of the 30th Euromicro Conference on Real-Time Systems (ECRTS)*, 2018.

[77] J. L. Herman, C. J. Kenna, M. S. Mollison, J. H. Anderson, and D. M. Johnson, "RTOS support for multicore mixed-criticality systems", *Proceedings of the 18th Real Time and Embedded Technology and Applications Symposium*, pp. 197–208, 2012.

[78] A. Paolillo, P. Rodriguez, V. Svoboda, O. Desenfans, J. Goossens, B. Rodriguez, S. Girbal, M. Faugere, and P. Bonnot, "Porting a safety-critical industrial application on a mixed-criticality enabled real-time operating system", *Proceedings of the 5th Workshop on Mixed Criticality Systems (WMC), RTSS*, pp. 1–6, 2017.

[79] C.-G. Lee, H. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim, "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling", *IEEE transactions on computers*, vol. 47, no. 6, pp. 700–713, 1998.

[80] R. Davis, S. Altmeyer, and A. Burns, "Mixed criticality systems with varying context switch costs", *Proceedings of the Real Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 140–151, 2018.

[81] A. Burns and R. I. Davis, "Adaptive mixed criticality scheduling with deferred preemption", *Proceedings of the 35th IEEE Real-Time Systems Symposium*, pp. 21–30, 2014.

[82] N. C. Audsley, "On priority asignment in fixed priority scheduling", *Information Processing Letters*, vol. 79, no. 1, pp. 39–44, 2001.

[83] A. Bertout, J. Forget, and R. Olejnik, "Automated runnable to task mapping", Tech. Rep., May 2013.

[84] E. Oklapi, M. Deubzer, S. Schmidhuber, E. Lalo, and J. Mottok, "Optimization of real-time multicore systems reached by a genetic algorithm approach for runnable sequencing", *Proceedings on the International Conference on Applied Electronics*, pp. 233–238, 2014.

[85] H. R. Faragardi, B. Lisper, K. Sandström, and T. Nolte, "An efficient scheduling of autosar runnables to minimize communication cost in multi-core systems", *Proceedings of the 7th International Symposium on Telecommunications (IST)*, pp. 41–48, Sep. 2014.

[86] International Standards Organisation, "26262 - road vehicles-functional safety", 2011.

[87] A. Burns, K. Tindell, and A. Wellings, "Effective analysis for engineering real-time fixed priority schedulers", *IEEE Transactions on Software Engineering*, no. 5, pp. 475–480, 1995.

[88] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: Exact characterization and average case behaviour", *Proceedings of the 10th Real Time Systems Symposium*, pp. 166–171, 1989.

[89] E. Bini and G. Buttazzo, "Measuring the performance of schedulability tests", *Real-Time Systems*, vol. 30, no. 1-2, pp. 129–154, 2005.

[90] S. Kramer, D. Ziegenbein, and A. Hamann, "Real world automotive benchmarks for free", *Proceedings of the 6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, 2015.

[91] L. Huang, I.-H. Hou, S. S. Sapatnekar, and J. Hu, "Graceful degradation of low-criticality tasks in multiprocessor dual-criticality systems", *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, pp. 159–169, 2018.

[92] A. Burns and S. Baruah, "Towards a more practical model for mixed criticality systems", *Proceedings of the 1st Workshop on Mixed-Criticality Systems, RTSS*, pp. 1–6, 2013.

[93] M. Hamdaoui and P. Ramanathan, "A dynamic priority assignment technique for streams with (m, k)-firm deadlines", *IEEE transactions on Computers*, vol. 44, no. 12, pp. 1443–1451, 1995.

[94] G. Bernat and A. Burns, "Combining (m/n)-hard deadlines and dual priority scheduling", *Proceedings of the 18th Real-Time Systems Symposium*, pp. 46–57, 1997.

[95] G. Bernat, A. Burns, and A. Liamosi, "Weakly hard real-time systems", *IEEE transactions on Computers*, vol. 50, no. 4, pp. 308–321, 2001.

[96] O. Gettings, S. Quinton, and R. Davis, "Mixed criticality systems with weakly-hard constraints", *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, pp. 237–246, 2015.

[97] R. Medina, E. Borde, and L. Pautet, "Availability enhancement and analysis for mixed-criticality systems on multi-core", *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1271–1276, Mar. 2018.