UNIVERSITY *of York*

This is a repository copy of *Industrial Application of a Partitioning Scheduler to Support Mixed Criticality Systems*.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/148617/

Version: Published Version

Proceedings Paper:

White Rose
university consortium
Universities of Leeds, Sheffield & York

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

Industrial Application of a Partitioning Scheduler to Support Mixed Criticality Systems

Stephen Law
Rolls Royce Control Systems, Birmingham, UK
forename.surname@Rolls-Royce.com

Iain Bate
The University of York, York, UK
forename.surname@york.ac.uk

Benjamin Lesage
Rolls Royce Control Systems, Birmingham, UK
The University of York, York, UK
forename.surname@york.ac.uk

—————— **Abstract** ——————

The ever-growing complexity of safety-critical control systems continues to require evolution in control system design, architecture and implementation. At the same time the cost of developing such systems must be controlled and importantly quality must be maintained.

This paper examines the application of Mixed Criticality System (MCS) research to a DAL-A aircraft engine Full Authority Digital Engine Control (FADEC) system which includes studying porting the control system's software to a preemptive scheduler from a non-preemptive scheduler. The paper deals with three key challenges as part of the technology transitions. Firstly, how to provide an equivalent level of fault isolation to ARINC 653 without the restriction of strict temporal slicing between criticality levels. Secondly extending the current analysis for Adaptive Mixed Criticality (AMC) scheduling to include the overheads of the system. Finally the development of clustering algorithms that automatically group tasks into larger *super-tasks* to both reduce overheads whilst ensuring the timing requirements, including the important task transaction requirements, are met.

1 Introduction

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) [28]. 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 paper shall consider the model presented in [28], that defines DAL-A as the highest criticality level and DAL-E the lowest. It is typically assumed that the amount of effort assigned to producing enough evidence to prove the correct operation of a software component is directly proportional to its DAL [32]. However, in practice it is still essential that lower DAL software operates as expected to deliver the desired customer capability. Put simply a task's criticality is not necessarily related to its "importance".

In accordance with DO-178C [28], where two components developed against different DALs are integrated in the same system, it is necessary to guarantee that high DAL components have temporal and spatial isolation from "unproven" low DAL components.

There are two forms of partitioning that must be employed [29]: 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. This paper is principally concerned with the former, however in order to develop a safety critical system it is vital to consider both together.

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 [1]. The 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 [3]. 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 [3].

Instead, this paper examines the potential for applying a pre-emptive MCS scheduling approach, available in the literature, to a real industrial safety critical avionics application. The application chosen for prototyping was a Rolls-Royce DAL-A avionics engine control system. This system was taken off a certified in-service application without modification or simplification. The primary aim was to investigate whether a set of less critical monitoring functionality could be executed on the same processor as the DAL-A control software whilst still meeting its certification requirements. In order to support these requirements a move to a preemptive system was required in order to allow the introduction of temporal partitioning protection. Therefore a secondary aim was to study the migration of the existing non-preemptive fixed priority scheduling approach [8] to using preemptive scheduling. This offered further advantages such as responsiveness, avoidance of high blocking terms, however at the expense of greater RTOS overheads and complexity.

This work considers AMC+, the Adaptive Mixed Criticality (AMC) [5] variant, which uses a simple mode change protocol and recovery point on an idle tick [31]. AMC+ is chosen as it is the simplest fixed-priority scheduling approach for MCS that allows the system to return to low mode after a switch to high-criticality mode.

The paper describes the challenges involved in certifying a real industrial system with AMC+ and as part of doing so three main contributions are made. The first contribution, in section 3, is a process that explains how the scheduler can be constructed that when combined with timing watchdogs allows AMC+ to be efficiently implemented whilst giving the necessary temporal isolation between high and low criticality tasks. In section 4, given a suitable RTOS and watchdog mechanisms, the standard AMC+ schedulability analysis is extended to account for the real overheads of the Rolls-Royce aircraft engine control system, referred to in this paper as a Full-Authority Digital Engine Controller (FADEC). The extensions form the second contribution of the paper.

The main contributions of the paper, in section 5, are: demonstrating how real world requirements of task dependencies and jitter requirements can be handled; and a task clustering mechanism based around the tasks' deadlines that mitigates the overheads of the system such that more individual tasks are schedulable. This includes adherence to tight

jitter requirements placed on certain tasks, as well as transaction requirements involving multiple tasks. Even though the work is targeting the control system, the benefits of the proposed clustering algorithms are also demonstrated on a larger-scale evaluation.

Sections 3, 4 and 5 are preceded by a Related Work section and followed by the conclusions.

2   Background

The purpose of this section is to consider related work to this paper, explain the standard system model for AMC+, the standard schedulability analysis for AMC+, and finally the existing Rolls-Royce control system's approach to scheduling.

2.1   Related Work

This section considers the related work on two topics: reducing the cost of RTOS overheads and MCS. The work on reducing the cost of overheads has two main approaches: firstly making the analysis less pessimistic and secondly reducing the actual overheads. In terms of the analysis pessimism, the majority of work has been in 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 [24]. The focus in this work is not reducing the pessimism of the analysis, although such approaches could be applied as part of future work, but instead reducing the size of the 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 work has looked at minimising the number of priority levels, e.g. [2], which 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 super-tasks). This approach is the same philosophy as adopted in AUTOSAR systems where runnables are grouped to form tasks as part of reducing overheads [11, 16, 26]. The approach however suffers from restrictions that prevent their application in the current case; they either ignore transactions between tasks [11], or require the possibility to duplicate tasks shared between transactions [16]. This paper identifies, for the system studied, that the most appropriate method was the latter of the two mentioned here, that is reducing the number of tasks scheduled by the RTOS. This paper extends this research by examining task sets incorporating task transactions and jitter requirements.

Vestal [32] was one of the first publications to consider the schedulability of a MCS. The work draws the comparison that the reliability of the Worst Case Execution Time (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.

Building off Vestal's work, Baruah et al. [5] introduced three models for Mixed Criticality Scheduling - partitioned criticality, Static Mixed Criticality (SMC) and Adaptive Mixed Criticality (AMC).

Partitioned criticality [5] 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 ensures a timing error in a low DAL task cannot affect the temporal requirements of a high DAL task, therefore requiring no run time monitoring. However, because the scheduler will always execute a high DAL task if one is ready, it makes it significantly
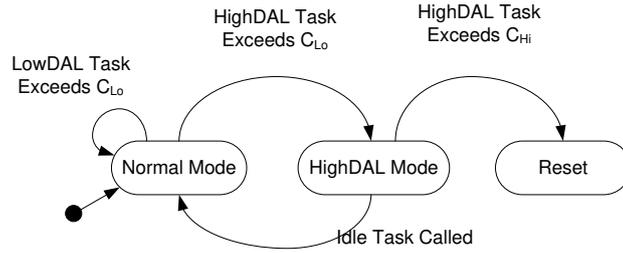
■ **Figure 1** AMC+ State Flow Diagram.

more difficult to meet low DAL task deadlines [5]. Furthermore as Baruah et al. [5] point out, the argument for avoiding run time monitoring is voided by the fact that many safety critical systems already incorporate run time monitoring for the purpose of error detection.

SMC and AMC [5] on the other hand assign task priorities according to their temporal requirements, regardless of criticality. The algorithms following Vestal's work assume that each task has a WCET bound (C_i^L) for each criticality level ($L = \{A, B, C, D, E\}$) where $C_i^A \geq C_i^B \geq C_i^C \geq C_i^D \geq C_i^E$. In most cases, previous works consider only two WCET bounds, C_i^{HI} and C_i^{LO}, where $C_i^{HI} \geq C_i^{LO}$. In this paper, DAL-A tasks have two execution times, C_i^{HI} and C_i^{LO}, and DAL-C tasks have just C_i^{LO}. Extensive software testing in representative engine operation simulations provide confidence that C_i^{LO} *should* not be exceeded, however this cannot be proved, and so C_i^{HI}, produced through WCET analysis, provides a safe upper bound.

SMC allows low or high DAL tasks to execute up to their C_i^{LO} or C_i^{HI} respectively; but they are then prevented from executing further [4]. This offers a stop dead point where any task must cease executing and provides adequate protection for high DAL tasks from low DAL 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 DAL tasks if *any* high DAL 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 DAL 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 [31]. The AMC+ protocol is achieved through a scheduler mode change which is summarised in Figure 1.

There are a great number of refinements to the AMC+ model, e.g. the Bailout protocol in [9,10], however these are largely independent to the contributions of this paper. That is, the papers lacked details of how the implementation would ensure the properties needed for certification, the analysis did not consider overheads as part of the analysis, and the papers did not consider how overheads could be reduced. Therefore, they are not covered here.

A number of papers have considered system development of an MCS. Sousa et al [30] identify the overheads induced by a multi-core task-split mixed criticality system, a number of the overheads identified and integrated into the schedulability analysis are similar to the overheads identified in this paper. However, in this paper we go further by presenting a full process for how overheads can be analysed and reduced.

Freitag et al [17] 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 [19] 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 [27] examine 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.

The focus of this paper is on applying an industrial avionics control system against the mixed criticality model originally presented by Vestal [32] as this provides a model well suited to industrial application. The specific implementation chosen is the AMC+ algorithm introduced by Baruah et al. [5]. The following sub-sections present more formally the baseline system model and schedulability analysis of AMC+.

2.2 System Model

A system is defined as a collection of tasks denoted by τ_i where $1 \leq i \leq N$. Each task τ_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 ($D_i \leq T_i$) if its execution must meet every deadline, whereas a soft deadline allows deadlines to be missed without having an adverse impact on the safe operation of the component.

Other parameters which describe a task include the release **jitter** J_i which denotes the maximum permissible variation of the period T_i for the release 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 τ_i to be reliant on the completion of another task τ_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 period of time.

Finally, a task τ_i is said to be schedulable if its **worst case response time (WCRT)** R_i, is less than its deadline D_i.

2.3 Static Schedulability Analysis

The static schedulability analysis assessed in this implementation follows the AMC-rtb algorithm introduced by Baruah et al. [5]. The schedulability analysis is performed in three stages. First, the response time of each task is assessed in the high and low modes. Finally, the response times of the high criticality tasks are assessed during a mode change from low to high.

The low and steady high mode response time analysis equations are defined below, where $hp(i)$ is the set of higher priority tasks than task τ_i and $hpH(i)$ is the set of high DAL higher priority tasks than task τ_i. Respectively $hpL(i)$ for the set of low DAL higher priority tasks. These equations calculate the high-criticality mode WCRT, R_i^{HI}, and the low-criticality mode WCRT, R_i^{LO}.

$$R_i^{LO} = C_i^{LO} + \sum_{j \in hp(i)} \left(\left\lceil \frac{R_i^{LO}}{T_j} \right\rceil C_j^{LO} \right) \tag{1}$$

$$R_i^{HI} = C_i^{HI} + \sum_{j \in hpH(i)} \left(\left\lceil \frac{R_i^{HI}}{T_j} \right\rceil C_j^{HI} \right) \tag{2}$$

Where each equation should be recursively solved. 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 [5] then defines the response time analysis for a high DAL task during a low-to-high mode change as follows:

$$R_i^* = C_i^{HI} + \sum_{j \in hpH(i)} \left(\left\lceil \frac{R_i^*}{T_j} \right\rceil C_j^{HI} \right) + \sum_{k \in hpL(i)} \left(\left\lceil \frac{R_i^{LO}}{T_k} \right\rceil C_k^{LO} \right) \tag{3}$$

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

2.4   Existing Control System

"Visual Fixed Priority Scheduler" (VisualFPS) is a task attribute assignment and scheduling analysis tool framework developed initially by Bate and Burns [8] 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 [8]. 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.

The current system is controlled using a non-premptive scheduler supported by a hardware timing watchdog. It consists of over 200 tasks, designed against an extensive set of system timing requirements [8]. Certification is achieved using timing information from a measurement-based timing analysis process [23] [22] [25].

The timing requirements for the task set include independent task requirements of period T_i, deadline D_i and in some cases completion jitter J_i as well as dependent task transaction requirements. 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. 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 [8].

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.

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 paper.

3 Achieving Sufficient Temporal Isolation

This section introduces the requirements surrounding the development of a MCS, both from a certification and a system integration point of view.

3.1 Certification Requirements

The paper focuses on the development of software for avionics systems. Accordingly, only the guidelines detailed in DO-178C [20] are explored in this section. However the guidelines are considered similar to those detailed in other software domains such as ISO26262 and IEC61508 [18].

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

These requirements form the basis for this mixed criticality assessment and they are explored further in the following sections.

3.2 Target Processor

The target processor for this analysis 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 [22,23]. These measurements are used in two ways. For High-DAL tasks the measurements are input into a hybrid measurement based WCET analysis tool and used for the task's C_i^{HI}. Secondly for both High-DAL and Low-DAL tasks the Maximum Observed Execution Time (or High Water Mark - HWM) obtained during the extensive software test regime is used as the task's C_i^{LO}.

Together with supervisor and user mode spatial partitioning control, this determinism provides the basis for an argument to be made that different software partitions executing upon the Rolls-Royce processor cannot affect each other from a spatial or temporal point of view.

3.3   Partitioning

The aim of this assessment was to identify how a set of low DAL monitoring tasks can be integrated alongside a set of high DAL-Control tasks. 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 DAL component, then a certification authority must assume a low DAL component is more likely to contain an error. So as guided by the requirements noted in Section 3.1, partitioning must be employed to prove that any errors that occur in a low DAL partition, cannot propagate to a high DAL partition. Partitioning is thus employed between high and low DAL tasks.

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 2 shows the statechart for the interrupt driven scheduler.

A timer driven interrupt is employed both to control the release of new tasks by invoking a scheduler tick, and to interrupt low DAL 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 DAL task the allowed execution time remaining.

High DAL tasks are not regulated in the same way. If a high DAL task executes beyond its C_i^{LO} then it is permitted to continue, but the next time the scheduler executes it will identify the need to move into the high DAL mode. This is controlled by the "Handle Overrun" block within Figure 2. Return to the low mode is controlled by a high DAL idle task.
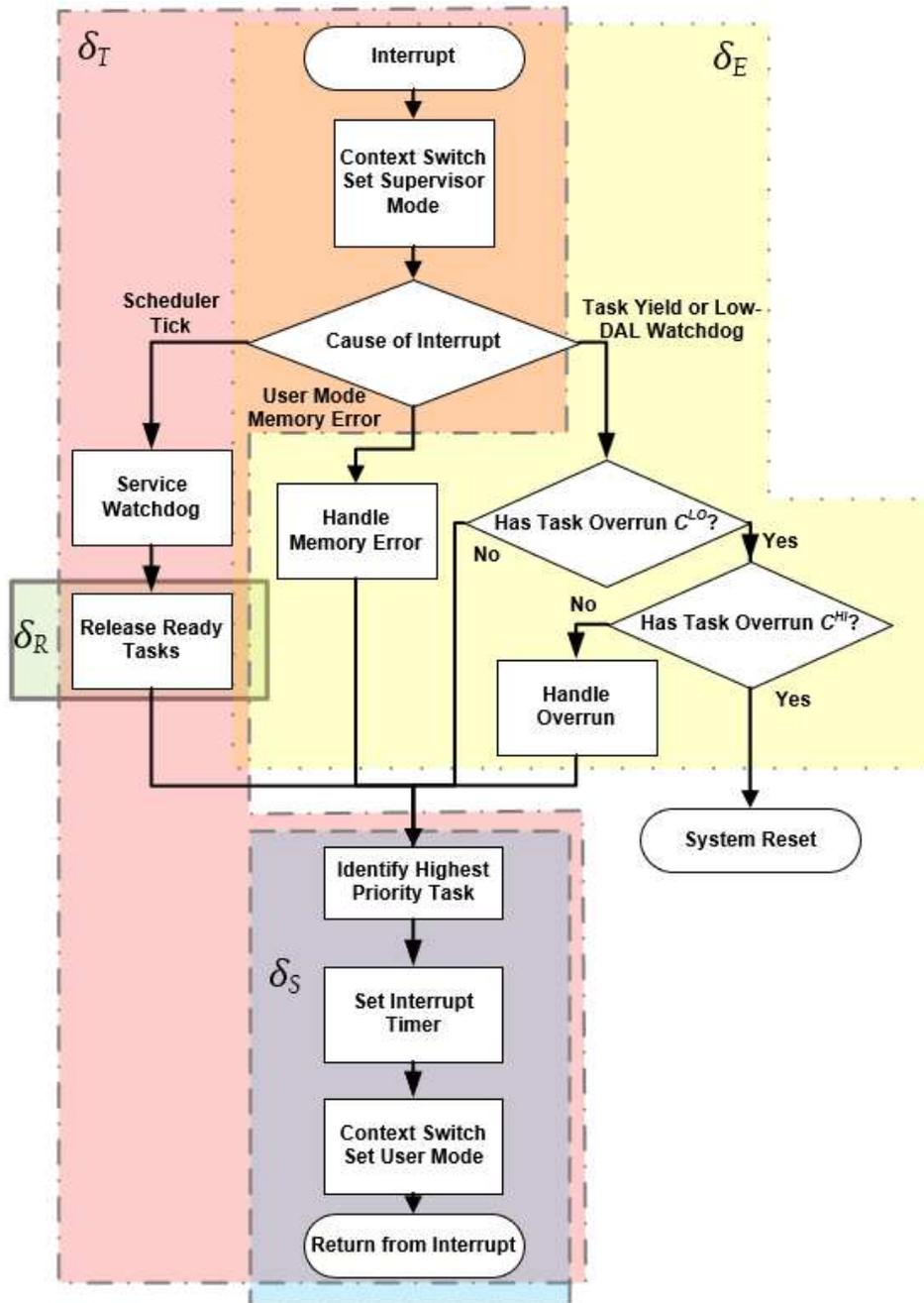
Secondly, processor memory protection is employed in a User/Supervisor arrangement. All tasks execute in a design-defined protected area of memory, with access to different hardware features or memory regions either permitted or restricted as necessary. Should any task execute outside these fixed boundaries, 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 2.

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 DAL tasks from low DAL task promiscuous memory or hardware interactions. Whereas the timer driven interrupt restricts the execution of low DAL tasks, ensuring a low DAL task cannot execute beyond its C_i^{LO}.

The processor is further protected by the *accepted and proven in use* hardware timing watchdog, which ensures any timing requirement not met is detected sufficiently quickly. Finally, at a system level the control system is designed around a full dual redundant architecture.

The earlier certification requirements are now re-visited with the reasons they are met in italics.

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
   The use of appropriately written and tested software makes this less likely, in addition the control system processor provides memory and other hardware protections through the use of a supervisor/user mode configuration. Furthermore, the processor is built upon a deterministic platform.

δ_I δ_E

Interrupt

Context Switch
Set Supervisor
Mode

Scheduler
Tick

Cause of Interrupt

Task Yield or Low-
DAL Watchdog

User Mode
Memory Error

Service
Watchdog

Handle
Memory Error

Has Task Overrun C^{LO}?

No

Yes

δ_R

Release Ready
Tasks

No

Has Task Overrun C^{HI}?

Handle
Overrun

Yes

System Reset

Identify Highest
Priority Task

δ_S

Set Interrupt
Timer

Context Switch
Set User Mode

Return from Interrupt

Figure 2 Partitioned Scheduler Statechart.

2. A partitioned software component should be allowed to consume shared processor resources only during its scheduled period of execution
   The software timing interrupt, controlled by the scheduler, should prevent this by bounding the execution time of low criticality tasks according to the task's 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.

3. Failures of hardware unique to a partitioned software component should not cause adverse effects on other partitioned software components
   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 a two-lane (duplex) architecture will ensure acceptable safety. Furthermore, the spatial partitioning employed shall ensure a task cannot interact with address regions outside of its permitted bounds

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 timing supervision component has been verified to DO-254 DAL-A and has been used on multiple certified systems.

4   Extending AMC+ Analysis to Allow for Overheads

In order to include the execution time of the scheduler shown in Figure 2 into the response time analysis for the system, the overheads were broken down into three constituent parts as described below:

1. Tick Overhead - δ_T (Figure 2 - dot/dash line, red). Calculated using the measured worst case tick time C_{TICK}. 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 C_{REL}. (Figure 2 - solid line, green)

2. Start Task Time - δ_S (Figure 2 - dashed line, blue). Calculated using the measured worst case task switch in time C_{START}. It includes:
   - 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 - δ_E (Figure 2 - dotted line, yellow). Calculated using the measured worst case task switch in time C_{END}.
   - The end time taken when a task finishes executing and returns to the scheduler.

Task releases are 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 ensures the maximum execution time for each overhead was captured firstly by ensuring the maximum number of tasks were in the released state at certain points, and that the maximum number of preemption points are observed.

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 proved to be linear against the number of tasks being released as per the design goal.

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

- The use of a time deterministic target processor.
- 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 [13]. This method of essentially viewing certain overheads as tasks provides a safe and suitable method for taking account of the periodicity of the overheads, and allows the overheads to be placed at the appropriate place in the schedule to ensure correct analysis of interference.

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

$$\delta_T^{MODE} = \left\lceil \frac{R_i^{MODE}}{T_{TICK}} \right\rceil C_{TICK} + \sum_{j \in MODE(i)} \left(\left\lceil \frac{R_i^{MODE}}{T_j} \right\rceil C_{REL} \right) \tag{4}$$

In Equation 4, as in the following Equations 5 and 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 DAL tasks will only occur in the low mode, or during a mode change from low criticality to high. This low DAL task release overhead cannot be ignored during a mode change because the release of these tasks occurs before the highest priority task begins to execute.

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

$$\delta_S^{MODE} = \sum_{j \in hpMODE(i)} \left(\left\lceil \frac{R_i^{MODE}}{T_j} \right\rceil C_{START} \right) \tag{5}$$

$$\delta_E^{MODE} = \sum_{j \in hpMODE(i)} \left(\left\lceil \frac{R_i^{MODE}}{T_j} \right\rceil C_{END} \right) \tag{6}$$

Equations 1 and 2 can therefore be extended as follows:

$$R_i^{LO} = C_i^{LO} + C_{START} + \delta_T^{LO} + \sum_{j \in hp(i)} \left(\left\lceil \frac{R_i^{LO}}{T_j} \right\rceil C_j^{LO} \right) + \delta_S^{LO} + \delta_E^{LO} \tag{7}$$

$$R_i^{HI} = C_i^{HI} + C_{START} + \delta_T^{HI} + \sum_{j \in hpH(i)} \left(\left\lceil \frac{R_i^{HI}}{T_j} \right\rceil C_j^{HI} \right) + \delta_S^{HI} + \delta_E^{HI} \tag{8}$$

Finally Equation 3 can be extended as follows:

$$R_i^* = C_i^{HI} + C_{START} + \delta_T^* + \sum_{j \in hpH(i)} \left(\left\lceil \frac{R_i^*}{T_j} \right\rceil C_j^{HI} \right) + \sum_{k \in hpL(i)} \left(\left\lceil \frac{R_i^{LO}}{T_k} \right\rceil C_k^{LO} \right) + \delta_S^* + \delta_E^* \quad (9)$$

This overhead model is built on the assumption that a switch from a low to a high DAL task, and the associated context switch, takes a constant time. On the deterministic Rolls-Royce aerospace company processor this is the case. However for less deterministic processors, or more complex context or thread switching mechanisms this model, and associated priority assignment mechanism, it may need to be adapted in a similar way to the work presented by Davis et al. [14].

This section has introduced an overhead model for calculating the response time of tasks in a preemptive mixed criticality system. The following section will now examine how these overheads can be minimised through appropriate system design.

5   Clustering to Reduce the Overheads of a Pre-Emptive Mixed Criticality System

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 (§ 2.4). The system follows a correct by construction design approach; each task's deadline is calculated as part of the design process to ensure that if all tasks meet their deadline (as confirmed through response time analysis), then the system as a whole will meet its temporal requirements. This relies on a task attribution assignment which calculates each task's deadline based on its period, transaction and completion jitter requirements. This system can then be said to comply with its timing requirements provided all tasks meet their deadlines.

The overhead assessment and implementation rules defined in Section 4 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 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 N is reduced to $N_{CLUSTER}$ (referring to Equation 4) reducing the overhead induced by task releases. Furthermore, the number of higher priority tasks $hpMODE(i)$ is reduced in Equations 5 and 6, reducing the start and stop task induced overheads.

This meant that simply porting the existing control system task set to the new architecture produced a system whose overheads exceeded 40% of the total system utilisation, making the system unschedulable. Therefore, it is necessary to condense the set of tasks in order to reduce the RTOS overheads to a level where a schedulable system can be defined.

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 paper is concerned principally with the latter, with the former being considered in parallel work.

A number of methods were investigated for controlling the clustering, all methods were defined using some or all of the temporal requirements placed on the tasks within a system. Each method follows the same basic process of placing tasks into a defined order, where

---

Algorithm 1 Task Clustering Algorithm.

1: */* Calculate task deadlines based on each task's timing requirements */*
2: UnOrderedTasks = CalculateTaskDeadlines(Period, Jitter, Transactions)
3: */* Create OrderedTaskSet according to the defined clustering method */*
4: **switch** (ClusteringMethod) **is**
5:   **case Period:**
6:     */* Order tasks based on their period, lowest period taking the highest priority */*
7:     OrderedTaskSet = OrderByPeriod(UnOrderedTasks)
8:   **case Transaction:**
9:     */* Order tasks based on their transaction requirements. Transactions with the lowest overall deadline take the highest priority */*
10:    OrderedTaskSet = OrderByTransaction(UnOrderedTasks[Task $\in$ Transactions])
11:    */* Order tasks without transaction requirements based on their jitter requirements first, and finally any remaining tasks by their period. */*
12:    OrderedTaskSet += OrderByJitter(UnOrderedTasks[Task $\notin$ OrderedTaskSet])
13:    OrderedTaskSet += OrderByPeriod(UnOrderedTasks[Task $\notin$ OrderedTaskSet])
14:   **case Jitter:**
15:    */* Order tasks based on their Jitter requirements. Prioritising tasks with small Jitter requirements */*
16:    OrderedTaskSet = OrderByJitter(UnOrderedTasks)
17:    OrderedTaskSet += OrderByTransaction([(Task $\notin$ OrderedTaskSet) $\in$ Transaction])
18:    OrderedTaskSet += OrderByPeriod([Task $\notin$ OrderedTaskSet])
19:   **case Deadline:**
20:    */* Order tasks based on their Deadline, shortest Deadline taking the highest priority */*
21:    OrderedTaskSet = OrderByDeadline(UnOrderedTasks)
22:
23: Move OrderedTaskSet[0] into SuperTask[0]
24: SuperTask[0].Period = OrderedTaskSet[0].Period
25: SuperTask[0].Criticality = OrderedTaskSet[0].Criticality
26:
27: j = 0
28: **for** i in 1..OrderedTaskSet.Length **do**
29:   **if** OrderedTaskSet[i].Period is NOT a harmonic of OrderedTaskSet[i-1].Period **or** OrderedTaskSet[i].Criticality $\neq$ SuperTask[j].Criticality **then**
30:     j++
31:     Move OrderedTaskSet[i] into SuperTask[j]
32:     SuperTask[j].Period = OrderedTaskSet[i].Period
33:     SuperTask[j].Deadline = OrderedTaskSet[i].Deadline
34:     SuperTask[j].Criticality = OrderedTaskSet[i].Criticality
35:   **else**
36:     Move OrderedTaskSet[i] into SuperTask[j]
37:     SuperTask[j].Deadline = min(OrderedTaskSet[i].Deadline, SuperTask[j].Deadline)
38:     SuperTask[j].Period = GreatestCommonDivisor(OrderedTaskSet[i].Period, SuperTask[j].Period)
39:   **end if**
40: **end for**
41:
42: Apply DMPO to SuperTask set

Algorithm 2 Extension to Support Deadline_D.

29: **if** Task[i].Period is NOT a harmonic of Task[i-1].Period
    or Task[i].Criticality $\neq$ SuperTask[j].Criticality
    or Task[i].Deadline $\neq$ SuperTask[j].Deadline

---

nominally the first task would form the highest priority, and the last task the lowest. The next step is to step through the ordered set breaking it down into super-tasks. The methods studied are introduced in Algorithm 1 and described below:

- No Clustering - no clustering is performed
- Period - Tasks were ordered according to period, then one super-task was formed for each different task period
- Jitter - Tasks with completion jitter requirements were first organised into super-tasks, before all remaining tasks were ordered and grouped using their transaction requirements (if applicable) and finally period.
- Transactions (Trans) - Tasks with cross-task transactional requirements were first placed into super-tasks along with the other tasks in their transaction. The remaining tasks were then sorted into super-tasks first using their jitter requirements (if applicable) and finally periods.
- Deadline_D (D_D) - Tasks were first ordered using their computed deadline, then the ordered set was divided into super-tasks across task period boundaries. Finally, tasks with different deadlines are not clustered together.
- Deadline_P (D_P) - As Deadline_D, however tasks with different deadlines were permitted to be clustered together.

In all cases no super-task contains tasks of different criticalities, this is vital to comply with the partitioning approach introduced in section 3.3. Finally, if any two tasks' timing requirements are identical, then their order is decided arbitrarily.

For brevity, Deadline_D is not shown explicitly in Algorithm 1. Unlike Deadline_P, Deadline_D does not cluster together tasks that have different deadlines. Deadline_D is implemented by replacing line 29 of Algorithm 2 with line 29 of Algorithm 1.

As the deadline of a task is derived as described in [7], this in one respect means the Deadline clustering methods are an implicit amalgamation of the Jitter, Transactions and Period clustering methods.

In all cases the deadline for each super-task was set to the lowest deadline of any task inside the super-task. The set of RTOS super-tasks were then prioritised using the DMP Oprotocol [8].

In addition to the above stated clustering methods a search based optimisation algorithm was used to attempt to identify an appropriate task set, however in no cases did it deliver improved results. The reason is the transaction requirements have to be carefully handled, which meant a guided random algorithm did not perform well. Therefore it is not included in this paper.

6 Evaluation & Results

The various approaches to task clustering and their impact on the RTOS overheads were assessed in three ways. Firstly, the clustering methods were each applied to a real Rolls-Royce DAL-A control system, which was then complimented using a set of DAL-C monitoring functions; thus providing a Mixed Criticality System for analysis. Secondly, the clustering

▪ **Table 1** Clustering Results When Applied to the Rolls-Royce Aircraft Engine Control System.

	#SuperTasks	Sched Tasks	Trans Pass?	δ_S	δ_E	δ_T	δ_{SUM}
NoClustering	228	24.1%	Yes	17.6%	21.1%	6.3%	45.0%
Period	17	85.5%	No	2.5%	3.0%	2.1%	7.6%
Transaction	10	9.2%	Yes	4.0%	4.8%	2.5%	11.4%
Jitter	53	38.2%	No	13.1%	15.7%	5.1%	33.9%
Deadline_D	167	40.4%	Yes	11.7%	14.1%	4.7%	30.5%
Deadline_P	15	100.0%	Yes	0.8%	0.9%	1.6%	3.3%

algorithms were applied to a publicly available aircraft engine control case study taken from [6]. Finally, a random task set generator was used to produce a large number of tasksets, each of which was clustered using the set of clustering algorithms.

The actual RTOS overhead figures used through the temporal analysis are as measured on the Rolls-Royce control system application, and were obtained during a system test campaign on an aircraft engine control system test rig, and are not discussed further in this paper. However, in order to ensure the analysis was not influenced by these figures, the study also investigated how the proposed methods responded to varying RTOS overheads.

6.1 "Current" Rolls-Royce Engine Control System Example

The Rolls-Royce control software 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 [23]. The HWM and WCET were used for the C_i^{LO} and C_i^{HI} respectively. Additional monitoring tasks were added to the system simulating DAL-C functionality, which increased the number of tasks in total to 228. The results from applying each clustering algorithm to the engine control system are shown in Table 1. The table shows the percentage of tasks whose worst case response time is lower than its deadline (Schedulable Tasks), whether all transactions have maintained the correct order (Transactions Fulfilled), and the utilisation of the RTOS.

Each task has a defined period, approximately 5% of tasks have completion jitter requirements and approximately 50% of tasks form part of a transaction. Each transaction requirement, which consists of between 2 and 11 tasks, reflects a specific execution, or priority, order that must be maintained to ensure compliance to system level timing requirements. These transactions are often interconnected, and can feature tasks with different periods. The task periods are taken from the set (2.5, 5, 10, 12.5, 25, 50, 100, 200, 500)ms.

The results in Table 1 showed that the *Deadline_P* clustering method was the only algorithm able to generate a schedulable system, this was in-spite of 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 super-tasks, 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.

Table 2 Clustering Results When Applied to an Aircraft Engine Control Case Study.

	#SuperTasks	Sched Tasks	Trans Pass?	δ_S	δ_E	δ_T	δ_{SUM}
NoClustering	71	94.4%	Yes	3.6%	4.3%	2.4%	10.4%
Period	5	53.5%	No	0.3%	0.3%	1.5%	2.1%
Transaction	3	59.2%	Yes	2.2%	2.6%	2.0%	6.8%
Jitter	8	38.0%	Yes	4.4%	5.3%	2.6%	12.3%
Deadline_D	33	97.2%	Yes	2.0%	2.4%	2.0%	6.4%
Deadline_P	2	100.0%	Yes	0.1%	0.1%	1.4%	1.5%

In comparison to the original system, this partitioned approach allowed low DAL 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 possible as the analysis was able to capitalise on the difference between each high DAL task's C_i^{LO} and C_i^{HI}.

6.2   Public Domain Engine Control System Example

The second case study used for this analysis has been taken directly from a publicly available aircraft engine control example, as documented in [6]. The task set features 71 individual tasks, conjoined by 24 different transactions. All tasks in the original system were schedulable with all transactions being met.

The results in Table 2 were similar to those produced by the Rolls-Royce case study in section 6.1, and showed that only the *Deadline_P* clustering algorithm was able to produce a schedulable system with all transactions being met. Similarly to the Rolls-Royce case study, the RTOS overhead was considerable lower with clustering than without, but in order to produce a fully schedulable solution it was necessary to prioritise *Transactions* and *Jitter* requirements equally, as performed by the *Deadline_D* and *Deadline_P* algorithms.

6.3   Random Task Set Generation Assessment

The random task set generator is based on a version of the UUniFast algorithm [12], 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. 1000 tests were then performed for each test configuration.

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 used in section 6.1.
- 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.
- 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 $C_i^{LO} \leq C_i^{HI} \leq 2C_i^{LO}$.
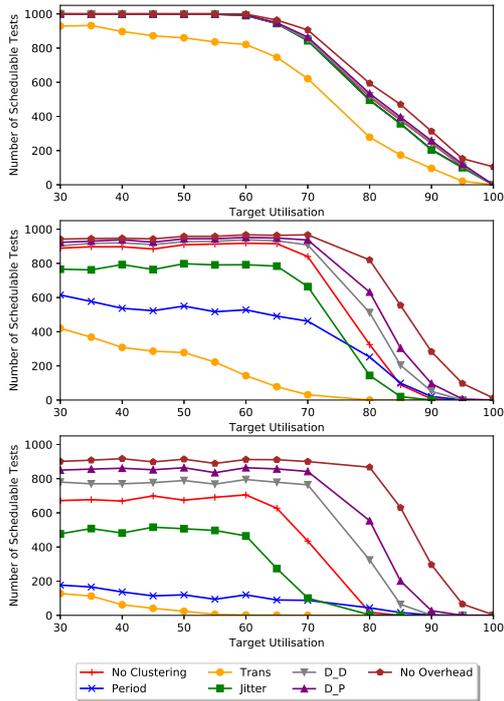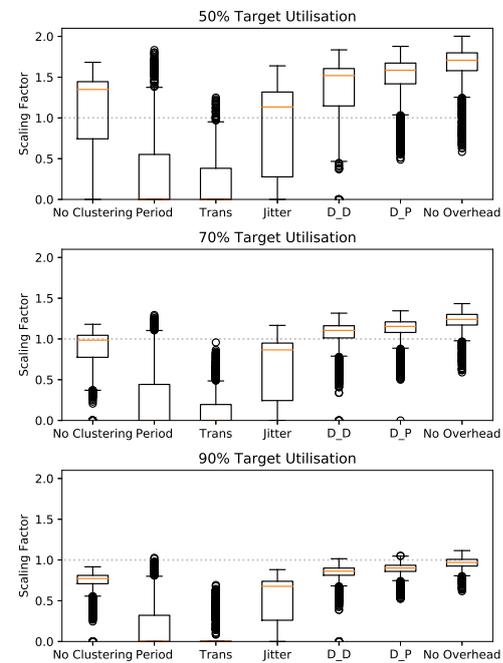
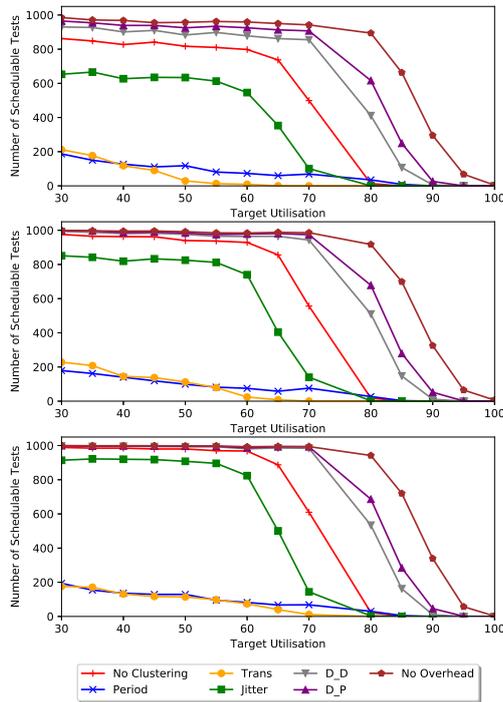**Figure 3** Schedulability of a 10, 50 and 100 Task System at Varying Target Utilisations.
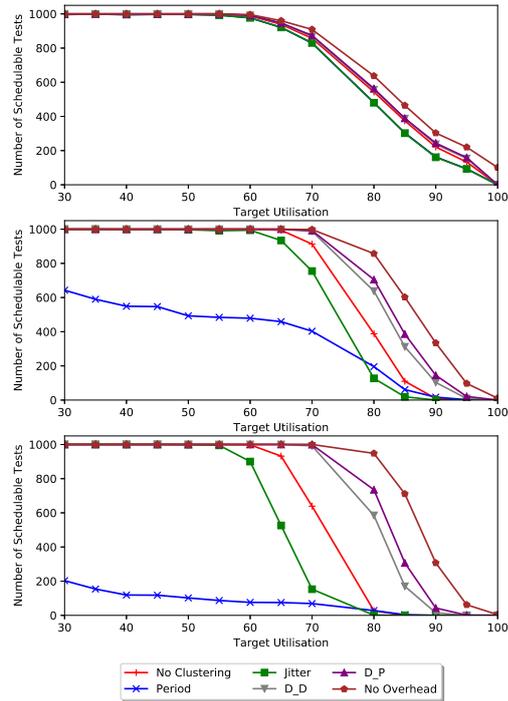
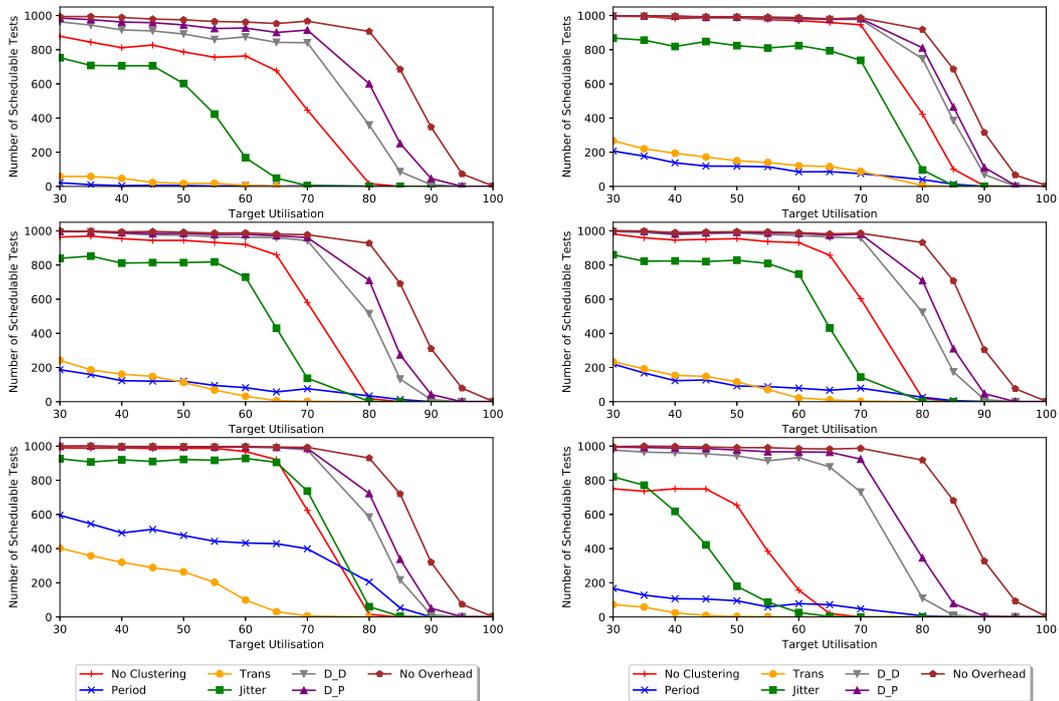**Figure 4** Schedulability of a 10, 50 and 100 Task System With No Transactions.
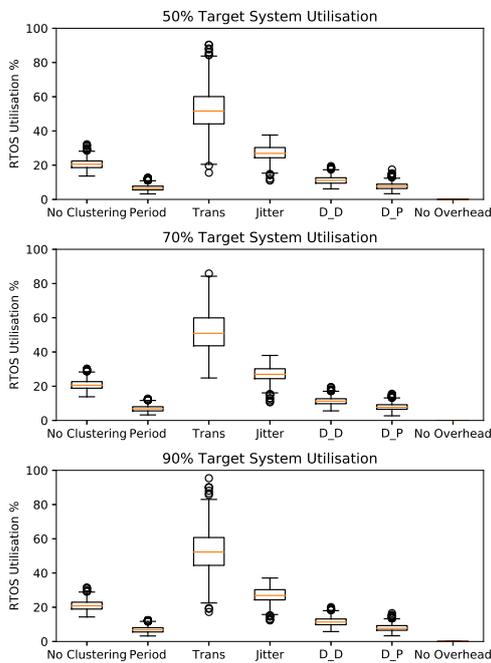
Figure 5 RTOS Overheads Calculated for each Clustered System.

Figure 6 Maximum WCET Scaling Factor to Provide a Schedulable System.

Figure 7 Number of Schedulable Tasks with Varying Transaction Rates [10%, 25% and 50%].

Figure 8 Number of Schedulable Tasks with Varying Jitter Rates of [0%, 5% and 10%].

Figure 9 Number of Schedulable Tasks with Low, Medium and High RTOS Overheads.

This follows principles similar to the approach defined in [21] where key characteristics are extracted from a real application and fed into a generator to derive representative benchmarks. However, the tasksets in [21] follow the AUTOSAR runnable model and do not include transactions which have a profound effect on the scheduling approach.

Figure 3 shows the number of schedulable tests out of the 1000 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 fails to take account of tasks with tight jitter requirements, which consequently receive lower priorities and fail their response time analysis.

For a system with 50 tasks, as shown in the second plot of Figure 3, 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 as it is able to minimise RTOS overheads by producing systems with less super-tasks. This hypothesis is further supported in Figure 5 which 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 able to achieve a 100% set of schedulable tests. This is because of the effect of transactions as shown by Figure 4 which shows the same test as shown by Figure 3, however without Transactions.

Figure 6 shows, for a 100 task system at varying target utilisation (50%,70%,90%), the analysed maximum possible WCET inflation factor, or sensitivity. 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. This figure is determined by sensitivity analysis. Therefore 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 5 to Figures 3 and 6; 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 shows that the aim of this clustering operation is not necessarily 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 7), varying jitter rates (Figure 8) and varying overheads (Figure 9). 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.

7   Conclusion

This paper has considered the application of a MCS scheduler to a DAL-A avionics engine control system. In particular, the paper examined how such a system should be developed and analysed to prove schedulability in the face of real life overheads. The partitioned architecture provides robustness against spatial and temporal infringements by low DAL tasks, and is capable of achieving a greater software utilisation than existing legacy systems.

The paper further considers the temporal requirements surrounding porting of an existing non-preemptive system. The study aimed to identify the most efficient implementation for porting a large system to a new pre-emptive Mixed Criticality System.

The work in this paper has targeted the AMC approach of [31], however the key contributions should be largely independent (or at least applicable to those that build off fixed priority scheduling) of which scheduling approach is used. The reasons are: the architectural approaches for detecting timing overruns and performing mode switches are independent of the policy; the overheads added to the *standard* analysis are only dependent on the architecture; and finally the method for reducing the number of pre-emptions and task releases is also independent of the scheduling approach. It is worth noting the benefits of the clustering algorithm would almost certainly be greater for systems with more complex processors as the preemption overheads would also include Cache-Related Preemption Delays.

For future work this study shall develop the implementation explored in this paper toward a full dynamic study of the operation and performance of the scheduled system. This will include assessing the quality of service provided to low DAL tasks and validating that a promiscuous low DAL task cannot affect the temporal performance of the system.

References

1   Airlines Electronic Engineering Committee. Avionics Application Software Standard Interface Part 1 - Required Services. *ARINC Specification 653 Part 1-3, Aeronautical Radio, Inc.*, 2010.

2   N. Audsley. On priority assignment in fixed priority scheduling. *Information Processing Letters*, 79(1):39–44, 2001.

3   N. Audsley and A. Wellings. Analysing APEX applications. In *17th IEEE International Real-Time Systems Symposium, (RTSS)*, pages 39–44, December 1996.

4   S. Baruah and A. Burns. Implementing mixed criticality systems in Ada. In *International Conference on Reliable Software Technologies*, pages 174–188. Springer, 2011.

5   S. Baruah, A. Burns, and R. Davis. Response-time analysis for mixed criticality systems. In *32nd IEEE International Real-Time Systems Symposium, (RTSS)*. IEEE, 2011.

6   I. Bate. *Scheduling and timing analysis for safety critical real-time systems*. PhD thesis, Citeseer, 1999.

7   I. Bate and A. Burns. An Approach to Task Attribute Assignment for Uniprocessor Systems. In *11th Euromicro Conference on Real-Time Systems*, pages 46–53, 1999.

8   I. Bate and A. Burns. An Integrated Approach to Scheduling in Safety-Critical Embedded Control Systems. *Real-Time Systems Journal*, 25(1):5–37, July 2003.

9   I. Bate, A. Burns, and R. Davis. A bailout protocol for mixed criticality systems. *IEEE Transactions on Software Engineering*, 2015.

10 I. Bate, A. Burns, and R. Davis. An enhanced bailout protocol for mixed criticality embedded software. *IEEE Transactions on Software Engineering*, 43(4):298–320, 2017.

11 A. Bertout, J. Forget, and R. Olejnik. Automated runnable to task mapping. Technical report, HAL, May 2013.

12 E. Bini and G. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

13 A. Burns, K. Tindell, and A. Wellings. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Transactions on Software Engineering*, 21(5):475–480, 1995.

14 R. Davis, S. Altmeyer, and A. Burns. Mixed Criticality Systems with Varying Context Switch Costs. In *Proceedings IEEE Real Time and Embedded Technology and Applications Symposium (RTAS)*, 2018.

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. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2018.

16 H. Faragardi, B. Lisper, K. Sandström, and T. Nolte. An efficient scheduling of AUTOSAR runnables to minimize communication cost in multi-core systems. In *7th International Symposium on Telecommunications (IST)*, pages 41–48, September 2014.

17 Johannes Freitag, Sascha Uhrig, and Theo Ungerer. Virtual Timing Isolation for Mixed-Criticality Systems. In *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

18 P. Graydon and I. Bate. Safety Assurance Driven Problem Formulation for Mixed-Criticality Scheduling. In *Proceedings of the Workshop on Mixed-Criticality Systems*, pages 19–24, 2013.

19 Jonathan L Herman, Christopher J Kenna, Malcolm S Mollison, James H Anderson, and Daniel M Johnson. RTOS support for multicore mixed-criticality systems. In *18th Real Time and Embedded Technology and Applications Symposium*, pages 197–208. IEEE, 2012.

20 B. Korel. Automated software test data generation. *IEEE Transactions on software engineering*, 16(8):870–879, 1990.

21 S. Kramer, D. Ziegenbein, and A. Hamann. Real world automotive benchmarks for free. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems*, 2015.

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

23 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*, 36(3):182–186, 2015.

24 C. Lee, H. Hahn, Y. Seo, S. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE transactions on computers*, 47(6):700–713, 1998.

25 B. Lesage, S. Law, and I. Bate. TACO: An industrial case study of Test Automation for COverage. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, RTNS '18, pages 114–124, 2018.

26 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. In *Proceedings of the International Conference on Applied Electronics (AE)*, pages 233–238. IEEE, 2014.

27 Antonio Paolillo, Paul Rodriguez, Vladimir Svoboda, Olivier Desenfans, Joël Goossens, Ben Rodriguez, Sylvain Girbal, Madeleine Faugere, and Philippe Bonnot. Porting a safety-critical industrial application on a mixed-criticality enabled real-time operating system. In *Proc. 5th Workshop on Mixed Criticality Systems (WMC), RTSS*, pages 1–6, 2017.

28 RTCA. Software Considerations in Airborne Systems and Equipment Certification. *DO-178C*, 2011.

29 J. Rushby. Partitioning for Safety and Security: Requirements, Mechanisms, and Assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999. Also issued by the FAA.

30 Paulo Baltarejo Sousa, Konstantinos Bletsas, Eduardo Tovar, Pedro Souto, and Benny Åkesson. Unified overhead-aware schedulability analysis for slot-based task-splitting. *Real-Time Systems*, 50(5-6):680–735, 2014.

31 K. Tindell and A. Alonso. A very simple protocol for mode changes in priority preemptive systems. Technical report, Universidad Politecnica de Madrid, 1996.

32 S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE International Real-Time Systems Symposium, (RTSS)*. IEEE, 2007.