A Practical Comparison of Scheduling Algorithms for Mixed Criticality Embedded Systems

Carl Cristian Arlock

Edward Linderoth-Olson

LUND UNIVERSITY

Department of Automatic Control

Msc Thesis
ISRN LUTFD2/TFRT--5959--SE
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Abstract

With the consolidation of automotive control processes onto single high-performance ECUs the issue of running, and thus scheduling, processes of varying criticality on a single CPU has moved to the fore. This has resulted in a number of new algorithms for scheduling such systems, for example Adaptive Mixed Criticality (AMC). This project attempts to measure the performance of some of these algorithms on a singlecore embedded system CPU and compares them in order to shed some light on their different advantages and disadvantages.

Keywords: Unicore, Scheduling, Embedded systems, Mixed Criticality, AUTOSAR, HCS12X.

Acknowledgements

We would like to thank Professor Gu Zonghua for the opportunity to do this work at the Department of Computer Science at Zhejiang University in Hangzhou.

Contents

1

Introduction

The migration of control systems from various scattered ECUs to fewer high-performace ECUs has brought forth the problem of having processes of varying importance to the system executing on the same CPU, and whereas it might not pose a serious problem if some were to miss their deadlines, others might have results bordering on the catastrophic. This manner of mixed criticality systems poses new challenges for scheduling processes, which have been met by an assortment of new scheduling algorithms. Many of these are very recent developments, and so little is known about their inherent advantages, disadvantages, and performance, in comparison to each other, something this project hopes to help rectify. In order to achieve this we have outlined the following research questions:

1. What scheduling algorithms are available for use in a multi criticality environment?
2. What is the performance of each of these measured in terms of generated overhead, response time and CPU utilisation?

The work has been performed for the most part at the Embedded System Engineering (ESE) Center at Zhejiang Univeristy in the People's Republic of China, under the tutelage of Professor Gu Zonghua. Research at the centre is primarily focused towards development of platforms for automotive physical information systems, standardised automotive electronics software development, automotive electronics simulation tests, and finally automobile intelligent environments. The RTOS used in this project, SmartSAR, has been developed at the centre. Parts of the project, primarily results processing and report writing, has been completed at the Department of Automatic Control at Lund university, guided by Professor Karl-Erik Årzén and Associate Professor Anton Cervin.

The work started with exploring viable algorithms. A real-time operating system was needed to run elicited algorithms and this had to be run on hardware with functionality and similar performance as in every day applications. Framework for generating a test system had to be implemented. Each elicited algorithm was implemented and fitted into the RTOS and then every test system generated had to be

Work	Author
Eliciting algorithms	Edward
Researching viability of using an Open Source RTOS	Carl Cristian
SmartSAR research	Edward, Carl Cristian
Test system generator framework	Edward
AMC test system generator implementation	Carl Cristian
EDF-DB test system generator implementation	Carl Cristian
ZSS test system generator implementation	Edward
AMC implementation	Carl Cristian
EDF-DB implementation	Carl Cristian
ZSS implementation	Edward
Result generation	Carl Cristian
Analysis of results	Edward
Report sections 2, 3.1, 3.2, 3.3, main parts of 4.1, 5.3, 5.4, 6	Edward
Report sections 1, 3.4, parts of 4.1, 4.2, 4.3, 4.4, 5.1, 5.2	Carl Cristian

Table 1.1 Individual contributions

run. From this, results were gathered and compiled for analysis. The work was done by Edward Linderoth-Olson and Carl Cristian Arlock. It was divided according to Table 1.1.

1.1 Outline

The report is divided into the following chapters

- Chapter two: Describes the background information and theory on which the report is based.
- Chapter three: How the work was performed and results gathered.
- Chapter four: The results gathered from the work.
- Chapter five: Analysis of the results.
- Chapter six: Includes conclusion and future work discussions.

2

Background

2.1 Introduction to real-time scheduling

An inherent problem in almost all real-world applications is the need for a program to perform several actions seemingly at the same time, perhaps regulating temperature and level in a water tank. In theory this could be handled by having a processor each for all the processes, but this is often impractical from a resource perspective. Instead the problem can be solved by the application of what is reffered to as 'time sharing concurrency', having the programmes execute in sequence but, through the use of special timing techniques, from a black box perspective it appears that they are executing concurrently [Årzén, 2011].

These timing techniques allowing the sharing of processor time, and other resources, are referred to as real-time scheduling. Perhaps an example would be useful to illustrate the concept in more detail. The process of producing chocolate ganache for danish pastries in a bakery requires both somewhat complicated temperature control and the pouring of a correct amount of cream into the chocolate, leading to two concurrent processes: one for the heat, and one for the cream. If the heat drops below 35 degrees centigrade the ganache will not mix properly and should the proportions of chocolate to cream be erronous the texture and taste will be off.

This gives us the system described in Figure 2.1. In this particular system, for the sake of simplicity, every time the flow of cream or the chocolate temperature needs to be corrected this takes an equal amout of cpu time and the processes that at that time uses the cpu runs until its finished what its doing. Every instance when the temperature or flow is calculated and corrected is referred to as a 'job', every process or task being comprised of a series of jobs. The deadlines, however, are slightly different for the two processes. The cream just needs to be finished before it is time to correct the flow again, since it is acceptable for the mixture to differ somewhat in its proportions, it can be rectified by adding somewhat less cream during the next iteration as long as the error is not too large. It is not even a catastrophe if the process should miss its deadline, as long as it does not do so too often. A process or system with these less stringent deadlines where a miss is occasionally tolerable is referred to as a soft real-time system [Årzén, 2011].

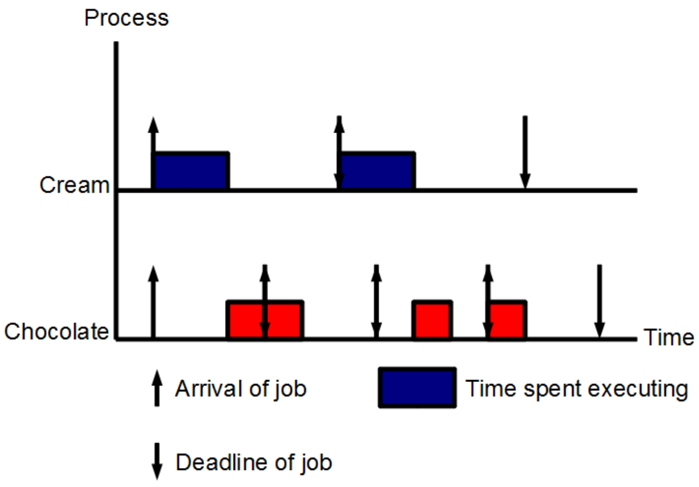

Figure 2.1 An example of two processes executing on the same procesor.

The chocolate on the other hand must never go below 35 degrees, and as the programme starts correcting the temperature very close to this value the deadline is much tighter, being based on when the temperature without appropriate action will drop too low. It is also critical that this deadline not be violated, as that will render the entire bach useless. A system with these strict deadlines where a miss is catastrophic is referred as a hard real-time system [Årzén, 2011].

The aforementioned system is quite content to run in this manner, each job only starting to execute when the last one is finished. However, had the execution time been somewhat longer for the cream process this would have caused the chocolate process to breach its deadline, as in Figure 2.2.

How then can this particular conundrum be solved? The acute observer might have noticed that the execution time between the arrival, or release as it is sometimes called, of the cream task until its deadline should be able to accomodate the execution of the chocolate process, but due to the latters' late arrival the cream task has already started executing and will doggedly continue to do so until it is finished. The solution is priorities. By assigning a higher priority to the chocolate task, its arrival will cause the cream process to be interrupted while it executes. The cream task can then continue once the chocolate process is finished, completing execution just in time to meet its deadline, see Figure 2.3.

Apart from the overarching problem of finding a schedule where all tasks meet their deadlines there are a number of more specific challenges associated with real-time scheduling, particularly when it comes to managing common resources. These problems arise when one or more tasks wish to access the same functions, e.g. memory, a screen or a thermometer. For example, two tasks can be 'deadlocked'

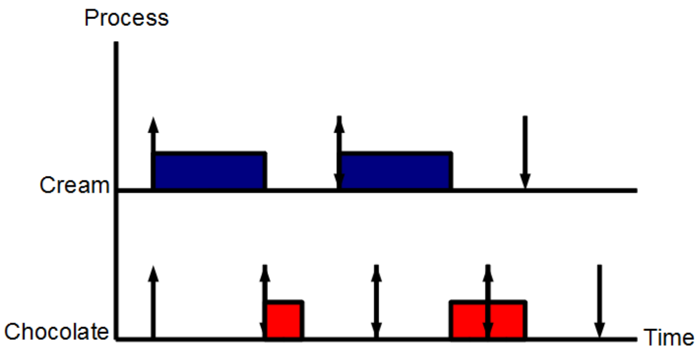

Figure 2.2 An example of two processes executing on the same procesor, with the chocolate process breaching its first deadline.

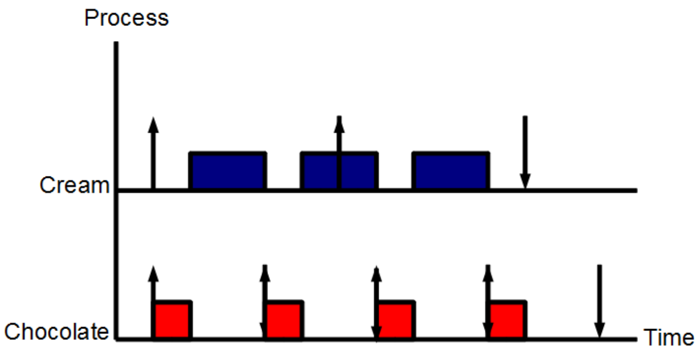

Figure 2.3 An example of two processes executing on the same procesor, with the cream process being pre-empted by the chocolate process.

waiting for the other to release a resource, or a lower priority task can accidentally block one of higher priority by locking a resource and being interrupted by one of middle priority that does not need the resource. However, as this project does not deal with shared resources, the investigation of this particular area is left to the reader's curiosity.

There are two ways of realising these real-time systems: a real-time programming language, e.g. Ada, or a sequential programming language augmented by a real-time operating system, e.g. the combination of C89 and SmartSAR used in this project [Årzén, 2011].

A real-time operating system (RTOS) can be likened to a bureacracy facilitating the use of the processor by the various programmes one wishes to run, providing the administrative framework for their operation. As such it should, although some civil servants will most likely disagree, be fast, small and efficient. Two of its most important functions that are at the center of this project are timing protection and context switching. The latter is one of the basic functions of any operating system, it refers to the process by which the memory housing all the variables and other paraphernalia belonging to the currently executing task is switched for those of the one next in line when it is time to hand over the processor. This allows the tasks to execute in their own environment – their context, hence the name [Årzén, 2011].

Timing protection is another basic function of any OS – the system that enforces and controls the deadlines, although its implementation can vary greatly. Once again it is the common resources that lie behind many of the difficulties. It might be intuitive that a task that has missed its deadline should be terminated immediately, however, should it be holding locks on shared resources these might not be correctly released. However, as mentioned previously, due to the nature of this project it is deemed to be outside the scope of this report.

There are a number of more stringent requirements on real-time operating systems (RTOS), setting them apart from traditional time-sharing operating systems like UNIX. The programmer and the processes must be able to perform actions they would normally not be allowed to, for instance suspending interrupts and changing the priorities of tasks. The system must also be extremely reliable, being able to run for a long time without restarting. It should also be quick to respond to I/O operations and able to complete those and other operations within a well-defined timespan [Årzén, 2011].

The chococlate ganache example relied on a method of scheduling referred to as pre-emptive fixed priority, i.e. processes can interrupt each other if they have a higher priority and priorities are fixed during execution. This is only one example of a large flora of algorithms used to schedule task sets, as demonstrated by the example depending on the configuration of the tasks a system may or may not be schedulable, which also means that a system that is schedulable with one task set might not be schedulable with another. One way to determine whether a certain algorithm can assure that the tasks in a task set do not break their deadlines is simply to calculate the schematic in the previous example, but in order for all the possible interference patterns to have been tested this must be done over the entire execution time for the programme, or the hyperperiod of the tasks. This quickly becomes exceedingly impractical. An easier method to determine the schedulability of a system is through the application of what is referred to as 'scheduling analysis'.

Scheduling analysis allows for the mathematical calculation of whether a certain system is schedulable under a certain algorithm, but as all algorithms are different there exists no one equation, each algorithm has its own requirements and conditions. There is one strict requirement for a schedulability condition; that it is a sufficient requirement for the system to be schedulable, i.e., if the condition indicates that the system is schedulable then all deadlines will be met. This can, however, mean that there are systems which are in fact schedulable but where the condition is too sceptical and indicates they are not, it is therefore desirable that the condition is necessary as well, i.e. that when it says the system is unschedulable there is always a situation where a deadline might be violated. If an algorithm's condition fulfills both of these requirements it is said to be exact.

This is, however, more readily understood through an example as well. The earlier used example of of the chocolate ganache does, however, due to the somewhat arbitrary priority setting, not confirm to any particular scheduling algorithm and as such lacks a readily available scheduling condition. Nevertheless, the rationale used is somewhat similar to that of the Rate Monotonic (RM) scheduling algorithm, a variant of pre-emptive fixed priority scheduling where priorities are assigned according to the frequency with which the tasks recur, with a shorter period entailing a higher priority. The chocolate process would thus have a higher priority just as earlier.

In order to determine whether the patissary task set is schedulable with RM the idea is to calculate the response time (R) for every task, i.e. the sum of the execution time (C) and the response time for all the jobs which can pre-empt it, and then compare this response time to the task's deadline (D). Should all the tasks in the system have response times shorter, or equal to, their deadlines then the system is schedulable. This results in a iterative equation running until the value for the tasks' response time stabilises, it is possible that the increasing response time due to interruptions from higher tasks in itself gives birth to even more interruptions. Equations 2.1 and 2.2 describe the initial setup for the example:

$$R_{chocolate} = C_{chocolate}, \tag{2.1}$$

$$R_{cream} = C_{cream} + \left\lceil \frac{R_{cream}}{T_{chocolate}} \right\rceil C_{chocolate}. \tag{2.2}$$

By inserting some values for the variables it is possible to iterate the equations to determine whether the system is schedulable. Note that in order for this paricular schedulability condition to be applicable the system must fullfil the following requirements:

- only periodic tasks
- deadlines and periods for a certain task must be equal
- no interprocess communication
- tasks may not suspend themselves
- priorities are unique
- the real-time kernel is 'ideal', i.e., context switches are performed instantly and no CPU time is lost to kernel overhead.

Using excution time 3, period (T) and deadline 5 for the cream process and exuction time 1, period and deadline 3 for the chocolate process this becomes:

$$R_{chocolate} = 1, \tag{2.3}$$

$$R_{cream} = 3 + \left\lceil \frac{0}{3} \right\rceil 1, \tag{2.4}$$

$$R_{cream} = 3 + \left\lceil \frac{3}{3} \right\rceil 1, \tag{2.5}$$

$$R_{cream} = 3 + \left\lceil \frac{4}{3} \right\rceil 1, \tag{2.6}$$

$$R_{cream} = 3 + \left\lceil \frac{5}{3} \right\rceil 1, \tag{2.7}$$

$$R_{cream} = 5. \tag{2.8}$$

As both response times are smaller than or equal to their respective deadlines the system is schedulable. Often, however, it is not this trivial. Large systems of tasks will make the calculation much more complex, and there are also some inherent problems in the variables themselves. Although the deadline is usually not too difficult to determine for a task, the execution time is generally impossible to get an exact value for; depending on which path the programme takes in its execution the result can vary greatly. As such programmers instead of exact values have to resort to statistical estimates with a certain margin of error, which in the unlikely event that they prove widely inaccurate for some special case can cause a task to exceed its allotted execution time leading in the worst case to deadline misses. The challenges associated with this are, however, discussed in more detail in the next chapter, on mixed criticality scheduling.

Apart from RM scheduling there are, as mentioned earlier, various other algorithms to schedule tasks depending on the demands of a particular system, one of the more common ones is Earliest Deadline First which simply schedules the job with the shortest upcoming deadline to run. Even so, a detailed discussion of this or any of the other algorithms is best left to the users discretion, as they are of limited use in the mixed criticality domain that is the focus of this paper.

2.2 Introduction to mixed criticality real-time scheduling

Mixed criticality scheduling is a relatively new discipline within software engineering, first introduced by Vestal's article [Vestal, 2007]. As previously mentioned it is born from the need to consolidate different processes of varying necessity, or criticality, to the system on a single CPU, hence the name. The problem derives from the nature of the worst case execution times (WCET) used for the scheduling calculations: most often it is impossible to determine an exact value for the WCET and the scheduler instead has to rely on an imperfect estimate. This estimate can be obtained in a wide variety of manners, sometimes a combination of different methods, but one can conjecture that as the imperativeness that the estimate hold true increases so also said estimates become more conservative. This threatens to give rise to a large amount of unutilised execution time for the normal case, which can differ significantly from the worst case [Vestal, 2007]. Another problem caused by this colocation of processes of mixed criticality on the same processor is criticality inversion - if the WCET limits are enforced stringently, a high criticality processes can be blocked when overrunning in order to let a lower criticality processes execute, whereas it might actually have been more desirable to let it continue until finished [Niz et al., 2009].

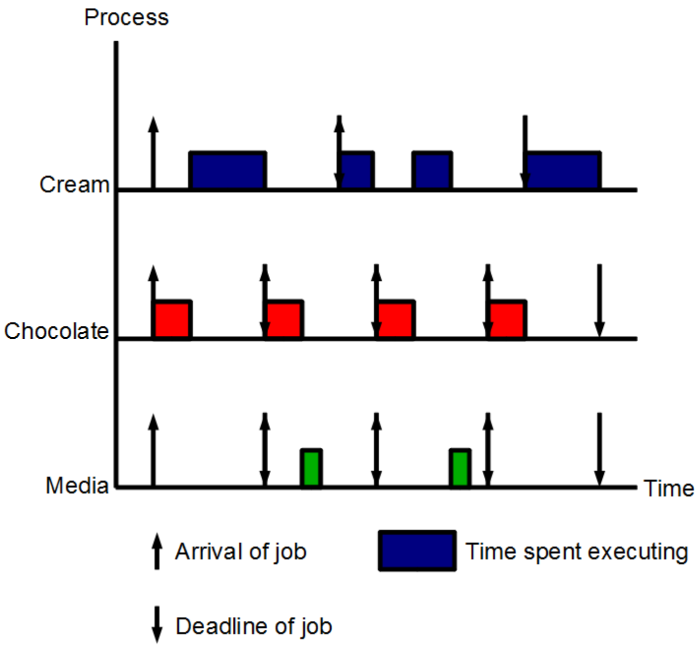

Figure 2.4 An example of two processes executing on the same procesor, assured at the criticality of the chocolate and cream processes. Note that the media process is blocked from executing in this example.

Perhaps, as with normal scheduling theory, mixed criticality scheduling is more readily understood through an example. Returning to our earlier patissary discussion of the chocolate ganache, let us asume that an overactive maintenance technician working at the bakery has decided that it is desireable to install a media player on the computer that runs the control process for the chocolate ganache. This will allow the bakery to play classical music and hopefully improve production as indicated in [Adrian North, 2001], provided cows are an adequate approximation of the bakery's staff.

However, a quick analysis indicates that it will be very difficult to include the media player process on the control computer using normal rate monotonic scheduling. In order to stop it from interfering with the more important tasks the technician gives the media player process the lowest priority, but this means it will be blocked from running by the other tasks and according to RM scheduling analysis the system is unschedulable, see figure 2.4. Even so, the technician suspects that the system should in fact be schedulable – due to the degree of safety necessary for the cream and chocolate process, their WCETs have been estimated quite conseravtively, and after performing some measurements it becomes apparent that the cream process is ususally much quicker than indicated by the WCET.

How then to get at this unutilised execution time? We introduce two levels of assurance that the system will not fail, a higher confidence one at the level previously used for the cream and chocolate processses, and a new lower one sufficient for the media player process. The acute reader might have realised that since the cream process was actually under a soft real time constraint whereas the chocolate process has a hard real time constraint, it might actually have been benificial to introduce three levels of criticality – separating the chocolate and cream processes. However, as the system has run excellently without this separation previously it only adds needless complexity, it is better to keep the solution simple, see [*Extreme Programming Pocket Guide* 2003]. Calculating the WCET estimates for the different levels of the criticality yields the results in Table 2.1.

Process	High criticality estimate	Low criticality estimate	Deadline
Cream	2	1.5	5
Chocolate	1	1	3
Media	1	0.5	3

Table 2.1 WCET estimates and deadlines for the different tasks on the bakery's control computer.

Now, given this lower level of assurance the system is actually schedulable, as in Figure 2.5. However, the probability of this latter system failing is higher than the one described by the high criticality measurements. How then to reconcile these two different depictions? There actually is a way, using scheduling analysis applied to these two different sets of measurements, that allows for the reconciliation of the higher criticality tasks' requirement of high assurance with the lower criticality tasks' disregard for it. Put simply, scheduling analysis is performed at the criticality of the task being examined, allowing each task's level of assurance to be set and evaluated individually. The equation for a task's reponse time thus becomes

$$R_i = C_i(L_i) + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j(L_i). \tag{2.9}$$

With priorities as in the previous example but with the media player process added as the lowest, WCETs (C) according to Table 2.1 and with the cream and chocolate processes having a high criticality (L) and the media player a low, the response time for the chocolate process thus becomes

$$R_{chocolate} = C_{chocolate}(L_{chocolate}), \tag{2.10}$$

$$R_{chocolate} = 1. \tag{2.11}$$

Respectively, the response time for the cream process becomes

$$R_{cream} = C_{cream}(L_{cream}) + \left\lceil \frac{R_{cream}}{T_{chocolate}} \right\rceil C_{chocolate}(L_{cream}), \tag{2.12}$$

$$R_{cream} = 1 + \left\lceil \frac{0}{3} \right\rceil 1, \tag{2.13}$$

$$R_{cream} = 1 + \left\lceil \frac{1}{3} \right\rceil 1, \tag{2.14}$$

$$R_{cream} = 1 + \left\lceil \frac{2}{3} \right\rceil 1, \tag{2.15}$$

$$R_{cream} = 2. \tag{2.16}$$

Finally, the response time for the media player process becomes

$$\begin{aligned} R_{media} = C_{media}(L_{media}) &+ \left\lceil \frac{R_{media}}{T_{chocolate}} \right\rceil C_{chocolate}(L_{media}) \\ &+ \left\lceil \frac{R_{media}}{T_{cream}} \right\rceil C_{cream}(L_{media}), \end{aligned} \tag{2.17}$$

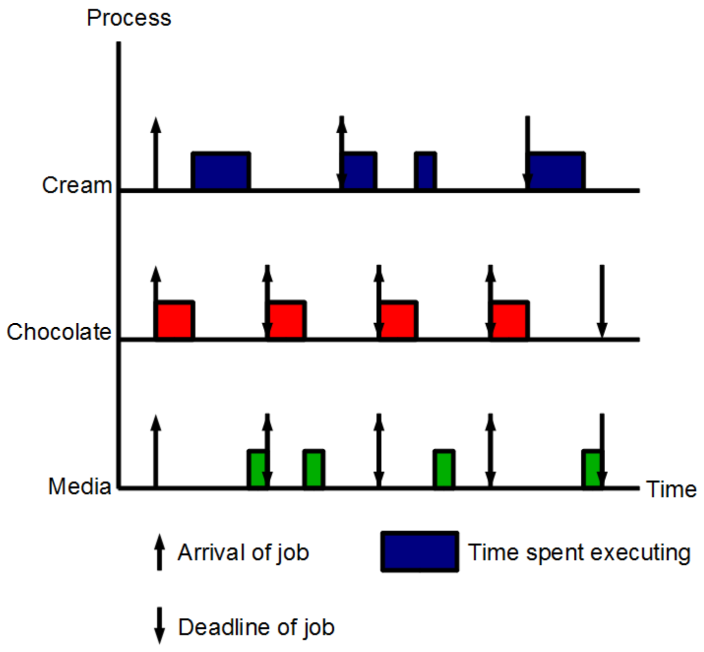

Figure 2.5 An example of three processes executing on the same procesor, asssured at the criticality of the media process.

$$R_{media} = 0.5 + \left\lceil \frac{0}{3} \right\rceil 1 + \left\lceil \frac{0}{5} \right\rceil 1.5, \tag{2.18}$$

$$R_{media} = 0.5 + \left\lceil \frac{0.5}{3} \right\rceil 1 + \left\lceil \frac{0.5}{5} \right\rceil 1.5, \tag{2.19}$$

$$R_{media} = 0.5 + \left\lceil \frac{3}{3} \right\rceil 1 + \left\lceil \frac{3}{5} \right\rceil 1.5, \tag{2.20}$$

,

$$R_{media} = 3. \tag{2.21}$$

All of which are within their respective deadlines, see table 2.1. The scheduling algorithm just applied is in fact one referred to as Static Mixed Criticality, or SMC for short. It is the predecessor of one of the algorithms discussed at length in this paper, Adaptive Mixed Criticality or AMC, and it is only one of a large number of different scheduling algorithms developed specifically to solve these manner of mixed criticality problems.

There are, however, a few points that should be noted and unfortunately are not readily apparent from the previous discussion. In order for SMC to work at least limited timing protection must be present – the RTOS must enforce WCETs for the tasks lest they overrun and cause criticality inversion. Criticality inversion is a phenomenon similar to priority inversion, a lower criticality task blocks a higher criticality one from executing, e.g. by executing for longer than its allotted budget. The reader might also have noticed that the high criticalty estimate for the media process was never actually used in the example. This is an advantage of SMC's, it only requires that tasks have their WCETs estimated up to and including their own level of criticality. It might not have been much of a boon in this particular example, but in systems with many low criticality tasks it can save a significant amount of resources as the process of estimating WCETs, as previously mentioned, is very expensive [Baruah et al., 2011].

In order to develop these algorithms and other tools to manage these problems a fair bit of research has been conducted into this area, resulting in a number of different algorithms. At the time of experimentation, autumn 2012, it was still very much an active field of research; the EDF-DB algorithm had been proposed in an article in July that same year. However, the field has continued to develop after the experiments where completed, for example Ekberg and Yi published som improvements to the EDF-DB algorithm [Ekberg and Yi, 2014]. The review of the field published by Alan Burns and Rob Davis, [Burns and Davis, 2013], covers the research area up until December 31st 2013 and provides an excellent overview of its development and scope until that date.

A more comprehensive discussion of the reasoning behind the different algorithms, their strengths and weaknesses, is left to Chapter 4.1.

3

Method

3.1 Scientific Method

The project was divided into three phases: a litereature study, implementation of the algorithms elicited by that study, and measurement of the above defined metrics. Apart from finding algorithms to implement the purpose of the literature study was to some extent to determine the results to expect for the different metrics. Also, it provided some guidance regarding the implementation choices that were made during the second phase, which are discussed in further detail in Section 3.2. In the last and final phase of the project the overhead was measured using a logic analyser, while the data for measuring utilisation and response times was provided by having termination messages printed to the systems COM port.

3.2 Implementation

For reasons of simplicity and to limit the scope of the project, we opted for a two-stage implementation: first in a fixed task priority (FTP) setting using only two algorithms, which we then extended with a third algorithm from the fixed job priority (FJP) domain.

The algorithms considered for implementation in an FTP system were:

1. Criticality As Priority Assignment (CAPA)
2. Own Criticality Based Priority (OCBP)
3. Period Transforming
4. Static Mixed Criticality (SMC)
5. Adaptive Mixed Criticality (AMC)
6. Zero-Slack Scheduling (ZSS)

CAPA has little merit other than as a benchmark, along with period transforming in conjunction with EDF, however, the benefit of such a benchmark system was deemed uncertain at best, and due to the limited resources available it was decided to focus on implementing an FJP algorithm instead. As OCBP is simply a variant of SMC that does not require runtime support implementing both is a little redundant, also since the latter is becoming more and more common according to [Baruah et al., 2011], SMC seems to be a more interesting choice although it will require slightly more work to implement than OCBP. On the other hand SMC is stricly dominated by AMC, which judging from the discussions in [Baruah et al., 2011] seems to in all manners be an improvement on SMC though slightly more difficult to implement. Zero-Slack Scheduling is a novel way of performing scheduling though it does require som patching from the original version, however judging by the experimentation described in [Huang, 2012] its performance is quite underwhelming. On the other hand, as the metrics considered for this project are different from that study it might still pose an interesting target for measurements. According to this reasoning, the first algorithms selected for implementation were ZSS and AMC.

For the second implementation phase the following algorithms were considered:

1. Priority List Reuse Scheduling (PLRS)
2. Earliest Deadline First - Virtual Deadlines (EDF-VD)
3. Earliest Deadline First - Demand Bound (EDF-DB)

PLRS is a FJP version of OCBP, however, again according to [Huang, 2012], it is quite slow and ponderous and as such not terribly relevant for the resource poor embedded systems setting. EDF-VD and EDF-DB are both new algorithms and are not included in Huang's study, so either would be interesting to implement from this point of view. However, seeing as EDF-DB is claimed to be an improvement over EDF-VD, it was decided to implement only this in order to limit the scope of the project.

3.3 Target Platform

The target architecture is a Freescale HCS12X board, running a variant of SmartSAR, a commercial AUTOSAR compliant RTOS developed by the Computer Science Department of Zhejiang University, which was modified to include code to measure execution and response times as well as overhead and CPU utilisation, and to support the different scheduling algorithms.

SmartSAR

SmartSAR is a compononent-based layered platform based on the AUTOSAR framework. SmartSAR is an implementation of the AUTOSAR framework which is a collaboration between major car manufacturers and other interests. The goal is to have a unified standard to increase reliability and portability. Individual software components will be thoroughly developed and specialised rather than being general purpose for fitting every type of operating system. The SmartSAR platform is comprised of the following components:

SmartSAR OS An OSEK compliant operating system that provides task scheduling, resource and event management and similar services to the RTE. [Li et al., 2009]

SmartSAR BSW For full AUTOSAR support the basic software module should support communication, memory and diagnostic services, however, in the SmartSAR BSW so far only the communication module is operational. [Li et al., 2009]

SmartSAR RTE The runtime environment supports applications, facilitates communications between them and allocates them to specific ECUs. It is also instrumental in making the applications platform independent, hiding implementation details of the OS and BSW systems. [Li et al., 2009]

SmartSAR HAL The hardware abstraction layer performs a similar function for the OS and BSW as the RTE for the applications, in making the OS and BSW more reusable. [Li et al., 2009]

SmartSAR IDE The integrated development environment contains tools for developing automotive electronics software, containing specific tools for modeling, configuration and code generation. [Li et al., 2009]

For this project the concern has primarily been the OS, and to some extent the RTE. The first as it contains the scheduling systems were the different algorithms were implemented, and the second as it was used by the Code Generator Java programme, see section 4.2, in order to generate the task systems used to test the algorithms. The latter, however, interfaced directly with the RTE's generated source files in order to create new systems, which renders the RTE's superstructure exceedingly irrelevant to this project. As such the SmartSAR OS will be examined in slightly more detail, whereas the others are left to the readers curiousity, e.g. by consulting [Li et al., 2009].

SmartSAR OS

As described earlier, SmartSAR is based on OSEK, specifically Smart OSEK OS. Its functions are provided to the RTE in the form of services, and can be grouped into the following main segments of functionality:

Task scheduling Scheduling, as explained in Chapter 2.1, comprises management of processor access between different tasks depending on their priorities and other qualifications. Basic SmartSAR supports three different strategies: non-preemptive, mixed preemptive and full preemptive scheduling. For the purposes of this project, full preemptive scheduling was used for all the algorithms. Significant changes were made to this module due to the nature of the project.

Event management Handles support for waiting for specific events to happen, this module was deactivated as there was no use for it in this project.

Alarm management Supports alarms capable of activating tasks, callback functions or setting events. In this project alarms were, among other things, used to implement the ZSI trigger function in the ZSS algorithm.

Resource management Facilitates sharing of common resources between tasks, or interrupts. This module was also deactivated for the purposes of this project.

Interrupt management Interrupts can be triggered by the OS to perform specific tasks, such as context switching. In SmartSAR there is support for two different variants of interupts, 1 and 2. Variant 1 is faster and can execute without triggering a context switch from the currently running task, however it cannot use OS system calls. Variant 2 is slower and more ponderous but can use OS system calls.

Stack monitor Supervises the stack during certain context switches, allowing early detection of stack overflow errors and the management thereof.

Timing protection Ensures that tasks do not exceed their documented worst-case execution times (WCET) and also monitors the time a task is blocked by lower priority tasks. This system has been partly disabled for some scheduling algorithms and heavily affected by the implementation of others.

Exception management Supports handling of exceptions to facilitate debugging and increase system robustness.

3.4 Simulated Load Generation

A program was written to generate a simulated load consisting of a variable number of tasks running in the system. Support for any number of tasks, which combined utilised a variable percentage of the CPU, was added. Default settings added a set number of tasks each with random run time and number of activations. Since run time and number of activations were uncertain for each task, the utilisation percentage for each set of tasks was not exact, only as close as possible to the chosen value. The program was able to generate sets of tasks for each algorithm with the three each requiring different preparations and precalculations. Each set ran for twenty seconds. For further discussion of this programme, please consult Section 4.2.

4 Results

4.1 Elicited Algorithms

Criticality As Priority Assignment (CAPA)

A very simple method for building an MC scheduler with very bad performance, it simply assigns criticality levels as priorities, it might however be useful as a benchmark.
Source: [Niz et al., 2009]

Period transforming

The simpler of two algorithms introduced by Vestal in his seminal article, the point is simply to transform the period of higher priority tasks to smaller intervals and then apply EDF scheduling.It is the basis for a few more advanced variants of EDF, EDF-VD and EDF-DB.
Source: [Vestal, 2007]

OCBP (SMC-NO)

A simple algorithm for assigning priorities so as to achieve an MC compatible system first introduced by Vestal. It does not handle exceeded WCETs, it just allows for different confidences in the estimations. The last article describes the algorithm when used to set the priorities of individual jobs (FJP), and not tasks (FTP), with varying priorities between these.
Source: [Vestal, 2007], [Li and Baruah, 2010]

Static Mixed Criticality (SMC)

Similar to OCBP, but a process cannot exceed its WCET. If it does, it is interrupted. As such, this algorithm requires runtime support. Also, this is primarily a FTP algorithm, although a port, as previously mentioned, of OCBP to FJP exists so in theory it should be possible to extend this in a similar manner.
Source: [Baruah et al., 2011]

Adaptive Mixed Criticality (AMC)

A successor to SMC, this seems a lot more useful. In this algorithm, as soon as a task exceeds its $C(LO)$, be it HI or LO criticality, all LO criticality tasks are suspended. Thus, this one also requires runtime support. This algorithm uses the previously established model with each task τ_i in a mixed criticality sporadic task set $\tau = \{\tau_1, ..., \tau_m\}$ with the following statistics

- $C_i(LO)$ and $C_i(HI)$ to denote worst case execution times in low- respectively high-criticality mode,
- D_i denotes the relative deadline,
- T_i is the period,
- L_i is the criticality.

AMC schedules tasks according to deadlines. The task with shortest deadline gets the highest priority. Accordingly the Audsley based Algorithm 1 sorts each task set according to deadline. This requires one priority per task, to sort the list. Each iteration, the task with the lowest deadline in the set is removed and placed in the list.

Schedulability is checked according to the calculations found in [Baruah et al., 2011] and is based on response time analysis. Method number two from the article was implemented and is discussed briefly here.

Based on the knowledge that a task τ_s invokes a criticality change at an arbitrary time s, an expression is derived. The task τ_i is impacted if the priority of τ_s is equal or greater than that of task τ_i. A formula for the response time of task τ_i is the following:

$$R_i^s = C_i(HI) + I_L(s) + I_H(s),$$

where $I_L(s)$ is the interference from low criticality tasks and $I_H(s)$ is the interference from tasks with higher or equal criticality:

$$I_L(s) = \sum_{j \in hpL(i)} (\lfloor s/T_j \rfloor + 1) C_j(LO).$$

For the interference of the HI criticality tasks, the formulas get more complicated. Here t denotes the time where task T_k interferes and $t > s$. The following function helps minimising the response time and is needed for calculation of interference from HI criticality tasks:

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

Interference term at time t becomes

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

where $R_i^* = \max(R_i^s) \forall s$. The values of s that have to be considered are in the interval $[0, R_i^{LO})$.

Algorithm 1 Audsley's algorithm

```
for each priority level $p$, lowest first do
    for each unassigned task $\tau$ do
        if $\tau$ schedulable at priority $p$ with all unassigned tasks assumed to have higher priorities then
            $\tau \leftarrow$ priority $p$
            BREAK to outer loop
        end if
    end for
end for
if set of unassigned tasks is empty then
    return schedulable
else
    return unschedulable
end if
```

Zero-Slack Scheduling (ZSS)

This algorithm is based on each task having two modes, Normal and Critical. When no slack (space for execution of lower criticality tasks) remains for a task to meet its deadline, the task switches from normal to critical mode, meaning that the execution of all lower criticality tasks is suspended. It was first put forward by de Niz et al in 2009, [Niz et al., 2009], however it was later revealed by Huang Huangming that this form failed to take into account one subset of task interference. Huang suggested a solution to this problem and thus the implementation in this project is based on his complementary work in [Huang, 2012].

Similarly to the response time calculation for RM scheduling used in the above example, ZSS uses a recursive algorithm to calculate the available 'slack' - the amount of idle processor time available for executing a task before its deadline. In order to account for the different manners of interference between tasks in a bi-modal algorithm such as this the calculations unfortunately become quite complex, as such it is perhaps best discussed in smaller parts at a time, from whence a whole shall emerge.

The crux of the problem is to find the Zero Slack Instant (ZSI), the point in time where there is so little slack available for a certain task, once interference from higher priority and higher criticality tasks has been taken into account, that it must switch to critical mode in order to meet its deadline. Switching to critical mode will suspend all lower criticality tasks in order to make their execution time available for the task. Should the zero slack instant become negative for any task, due to the response time being longer than the deadline, the task system is unschedulable. Algorithm 2 describes the algorithm used to find the ZSI, followed by descriptions of the functions used in that algorithm.

Algorithm 2 Calculation of task τ_i's zero slack instant

```
$s \leftarrow 0$
repeat
    $s' \leftarrow s$
    $C_i^c \leftarrow \max(C_i^0 - slack, 0)$
    $k \leftarrow K(C_i^c, D_i, \Gamma_i^c, \Delta_i^c(\xi))$
    $Z_i \leftarrow \max(D_i - k, 0)$
    $s \leftarrow S_i^n(\Gamma_i^n, Z_i)$
until $s = s'$ or $Z_i = D_i$
return $Z_i$
```

The K function displayed in Equation 4.1, calculates the minimum time required for the task τ_i to execute for the time t, where u is an upper limit to the execution time. The $\Delta_i(\Gamma, t)$ function returns the time demand generated by a taskset Γ for the duration t after the release of a job from τ_i,

$$K(t, u, \Gamma, \Delta_i) = \min\{\{u\} \cup \{t' \geq t | t' = t + \Delta_i(\Gamma, t)\}\}. \tag{4.1}$$

However, as the algorithm is bimodal two different variants of the Δ function prove necessary; one for τ_i's normal execution mode and one for when τ_i enters critical mode. These two functions are described in Equation 4.2 and Equation 4.3 respectively,

$$\Delta_i^n(\zeta_m, \Gamma, t) = \sum_{\tau_j \in \Gamma} \delta_i^c(\zeta_m, \tau_j, t), \tag{4.2}$$

$$\Delta_i^c(\zeta_m, \Gamma, t) = \sum_{\tau_j \in \Gamma} \delta_i^n(\zeta_m, \tau_j, t). \tag{4.3}$$

In both of these cases, the function for the task set Γ's time demand is of couse reliant on the time demand of each individual task τ_j in the set, calculated by the functions in equations 4.4 and 4.5,

$$\delta_i^n(\zeta_m, \tau_j, t) = \left(1 + \max\left(\left\lceil \frac{t - \phi_j^{i,n}(\zeta_m)}{T_j} \right\rceil, 0\right)\right) I_j^i(\zeta_m), \tag{4.4}$$

$$\delta_i^c(\zeta_m, \tau_j, t) = \left(1 + \max\left(\left\lceil \frac{t - \phi_j^c(\zeta_m)}{T_j} \right\rceil, 0\right)\right) I_j^i(\zeta_m). \tag{4.5}$$

As the reader with an eye for detail will quickly have noted, the difference between these two functions resides solely in the interference phase shift function ϕ – depending on whether the system is running in critical or normal mode the tasks will interplay differently, giving different values for the phase shift that generates the worst possible inteference. However, before delving into the details of the respective variants of that function, there is one common function used by both δ_i^n and δ_i^c that merits explanation: the function for the execution time of τ_j that will interfere with the execution of τ_i calculated at the criticality ζ_m, $I_j^i(\zeta_m)$, described as follows:

$$I_j^i(\zeta_m) = \begin{cases} \max(C_j(\zeta_m) - \theta_j(\zeta_m), 0) & \text{if } \tau_j \in L_i^{\zeta > \zeta_i}, \\ C_j(\zeta_m) & \text{otherwise.} \end{cases} \tag{4.6}$$

As is customary C_j denotes the execution time of task τ_j, however, with the added qualifier that it has been measured at the criticality level ζ_m; as mentioned previously execution times can be measured with varying degrees of assurance depending on the criticality level, and as such the execution time for a task will vary depending on at what criticality it has been measured. θ, on the other hand, is not so straight forward; it constitutes the minimum amount of slack that can be used by a task τ_i before it reaches its zero slack instant Z_i, calculated at a criticality level ζ_m. This is calculated as described in Equation 4.7, which will in fact render the algorithm recursive with the base case being when there are no further tasks that can interfere with task τ_i,

$$\theta(\zeta_m) = \max(Z_i - \Delta(\zeta_m, \Gamma_i^n, Z_i), 0). \tag{4.7}$$

Returning once again to the phase funtions ϕ^n and ϕ^c their individual construction is quite different although their function is the same, and as such they are perhaps best discussed in sequence. Starting with the phasing for the normal time demand function, it is calculated as described in Equation 4.8 and reliant on the delay between the release of task τ_i and the completion of task τ_j, denoted $r_j^{i,n}$. This once again makes the algorithm recursive, as is clear from Equation 4.9 $r_j^{i,n}$ is calculated using a special configuration of the K function, which was where this explanation originally started,

$$\phi_j^{i,n} = r_j^{i,n}(\zeta_m) + T_j - D_j, \tag{4.8}$$

$$r_j^{i,n}(\zeta_m) = K(C_j(\zeta_m), D_i, H_j^{\zeta \geq \zeta_i}, \Delta_i^n(\zeta_m)). \tag{4.9}$$

Moving on to the phasing of the critical mode time demand function its calculation differs significantly from that of the normal mode phasing, relying instead on the response time of the task τ_j as seen in Equations 4.10 and 4.11. However, just as in the calculation $r_j^{i,n}$, the equation hinges on the K function rendering it recursive until all interference has been accounted for,

$$\phi_j^{i,c} = C_j(\zeta_m) + T_j - r_j^{i,c}(\zeta_m), \tag{4.10}$$

$$r_j^{i,c}(\zeta_m) = Z_j + K(C_j(\zeta_m) - \theta(\zeta_i), D_j - Z_j, \Gamma_j^c, \Delta_j^c(\zeta_m)). \tag{4.11}$$

With the normal time demand function available, it is now possible to implement a function to calculate the available slack in a time period t; the function $S_i^n(t)$. As τ_i cannot make use of slack before its release time it becomes necessary to check all possible smaller intervals t' in the interval t,

$$\begin{gathered} S_i^n(t) = \max(t' - \Delta_i^n(\zeta_i, \Gamma_i^n, t')) | (\forall t' < t) \cup \\ (\forall t \geq t \text{where } \Delta_i^n(\zeta_i, \Gamma_i^n, t') = \Delta_i^n(\zeta_i, \Gamma_i^n, t))). \end{gathered} \tag{4.12}$$

Thus, all the parts necessary to execute algorithm 2 are assembled. The algorithm should be applied to the tasks in descending criticality-priority order, that is first all tasks with the highest criticality and priority, then all tasks with the highest criticality and second highest priority and so on and so forth. This is to ensure that the recursion through the tree of interference dependencies is done properly; it is reliant on the values for the interfering (higher priority and/or criticality) having already been calculated.

PLRS

An adaptation of FJP OCBP with quicker priority calculation. It is somewhat dubious as to how fast this really is, it seems it is more a case of it being less bad than the standard algorithm than actually good, which is corroborated by Huang's disputation.
Source: [Guan et al., 2011]

EDF-VD

A relatively new algorithm compared to the others, it seems no too difficult to implement (though it requires runtime support) and has not been compared to any others experimentally except normal EDF in a resource reservation setting. It based on a combination of EDF and period transforming, similar to what was introduced by Vestal, the novely of this algorithm lies in the way the new deadlines are calculated.
Source: [Baruah et al., 2012], ["Preemptive uniprocessor scheduling for mixed-criticality systems"]

EDF-DB

Another new algorithm based on EDF and period transformation, this utilises yet another way to calculate the new deadlines: it is based on demand bound functions. On a side note, as the algorithm is so new it is not included in Huang's disputation. This algorithm uses the previously established model with each task τ_i in a mixed criticality sporadic task set $\tau = \{\tau_1, ..., \tau_m\}$ with the following statistics

- $C_i(LO)$ and $C_i(HI)$ denote the worst case execution times in low- respectively high-criticality mode,
- D_i denotes the relative deadline,
- T_i is the period,
- L_i is the criticality.

As a task set is produced, every task is evaluated. Each task has their original deadline to begin with. Tuning a new, relative deadline according to the task set as a whole one can achieve schedulability in a previously unschedulable task set. This tuning is based on the demand bound functions introduced in [Baruah et al., 1990]. The basic principle is that if an upper demand of execution can be found on every task, including both LO and HI criticality tasks, the task set can be investigated to adjust the deadlines to achieve schedulability.

The demand bound functions are calculated for each specific task and criticality mode and gives an upper bound of the maximum execution time on any time interval of a given size. The formulas for calculating these functions have been omitted here and can be found in [Ekberg and Yi, 2012].

Algorithm 3, as published in [Ekberg and Yi, 2012], was implemented in the simulation support system to perform demand bound calculations on the task systems, iteratively tuning low-criticality relative deadlines.

In Algortihm 3, conditions A and B refer to the schedulability of the current task in LO and HI criticality mode respectively.

4.2 Implementation Specifics

Some of the results are very much implementation specific, but they show the difference between the three chosen algorithms. Even if they are implementation specific, it is clear that they work for the intended platforms.

While EDF-DB is time consuming to prepare, the schedulability rate, as is investigated in [Ekberg and Yi, 2012], is very high. The same goes for AMC. This is not the case with ZSS. While generating systems with a utilisation higher than 60%, increasing the amount of tasks proved to produce less and less schedulable systems. That is why all graphs lack data for more than 30 tasks in the ZSS systems – there where simply not enough schedulable systems.

Algorithm 3 EDF-DB algorithm

```
$candidates \leftarrow \{i | \tau_i \in HI(\tau)\}$
$mod \leftarrow \perp$
$l_{max} \leftarrow$ upper bound for $l$ in Conditions A and B
loop
    $final \leftarrow True$
    for $l = 0, 1, ..., l_{max}$ do
        if $\neg A(l)$ then
            if $mod = \perp$ then return FAILURE
            end if
            $D_{mod}(LO) \leftarrow D_{mod}(LO) + 1$
            $candidates \leftarrow candidates \setminus \{mod\}$
            $mod \leftarrow \perp$
            $final \leftarrow False$
            BREAK
        else if $\neg B(l)$ then
            if $candidates = \emptyset$ then return FAILURE
            end if
            $mod \leftarrow \arg\max_{i \in candidates}(dbf_{HI}(\tau_i, l) - dbf_{HI}(\tau_i, l-1))$
            $D_{mod}(LO) \leftarrow D_{mod}(LO) - 1$
            if $D_{mod}(LO) = C_{mod}(LO)$ then
                $candidates \leftarrow candidates \setminus \{mod\}$
            end if
            $final \leftarrow False$
            BREAK
        end if
    end for
    if $final$ then return SUCCESS
    end if
end loop
```

The other limit at 40 tasks is set by the limited amount of memory on the target hardware, there was simply not enough memory to fit more tasks in the system.

SmartSAR

SmartSAR is mainly written in C89 and auto generated code is added by the IDE. A basic system was initially created with the IDE. From then on, the Code Generator, described in Section 4.2, provided all code for the task system specific components like tasks, alarms, priority levels and so forth.. Most of the modifications were made in the operating system part of SmartSAR. The existing structures of the OS were used as far as possible. The resource management system was deactivated along with the multiple application support and event management features to allow changes to the scheduling functions. Variables for external and internal use were added to provide performance measurements and multiple criticality support. The main scheduling functions were identified and modified according to the needs of each scheduling algorithm. In the end result, without too many changes to the existing structures three different functions had to be modified to change the way the system handled scheduling. `Find_Next_Task`, `Insert_Ready_Task` and `Remove_Ready_Task` were called for scheduling needs, where the first was called at every system return from interrupts.

AMC

The AMC scheduling algorithm was implemented with support for two levels of criticality. From [Baruah et al., 2011], the following set of rules were implemented:

1. There is a *criticality level indicator* Γ, initialised at LO.
2. While ($\Gamma \equiv LO$), at each instant the waiting job generated by the task with highest priority is selected for execution.
3. If the currently-executing job executes for its LO-criticality WCET without signalling completion, then $\Gamma \leftarrow$ HI.
4. Once ($\Gamma \equiv$ HI), jobs with criticality level $\equiv$ LO will *not* be executing. Henceforth, therefore, at each instant the waiting job generated by the HI-criticality task with the highest priority is selected for execution.
5. An additional rule specifying that Γ be reset to LO if there are no HI-criticality tasks left to run.

The system was running a regular priority based FTP scheduler with these modifications to support multiple criticality scheduling. Tasks are put in a ready list according to priority and regular polling finds which task with highest priority that is ready to run.

The complexity of inserting, removing and finding next task is always O(1). Priority was assigned to each task using Audsley's algorithm and as such, there is one priority level per task. It follows that there is one ready list per task in this particular implementation and this increased memory utilisation.

Running Audsley's algorithm to assign priorities did not consume a significant amount of time. These were the only preparatory calculations in AMC and proved to be the fastest of the three implemented algorithms.

EDF-DB

The EDF-DB scheduling algorithm was implemented with support for two levels of criticality. The existing priority based FTP scheduler was removed and a new FJP scheduler was implemented. This implementation used a linked list to sort the tasks according to nearest deadline. The task with nearest deadline is always first in the list. In a worst case scenario, inserting into the list will be O(n). Removing and finding next task is O(1). This algorithm did not require more than one list and as such, the memory requirements are lower than those of AMC and ZSS.

This algorithm requires a lot of preparatory work. As noted in [Ekberg and Yi, 2012] the calculations to tune to the demands of the tasks requires at most

$$\sum_{\tau_i \in HI(\tau)} (D_i - C_i(\mathrm{LO}) + 1)$$

outer loop iterations and $l_{max} + 1$ inner loop iterations, where l_{max} is the system run time or hyperperiod, whichever is shorter, in ticks.

The large number of iterations in the preparatory calculations for EDF-DB took a lot of time. Even with a run time of only twenty (20) seconds as the amount of tasks increased the time to compute increased exponentially. The preparatory calculations of EDF-DB were by far the most time consuming.

ZSS

The implementation of the ZSS system can be divided into two parts: firstly, a system for calculating the tasks' zero slack instances (ZSIs) and secondly run time support for the mode switch and management of tasks exceeding their deadlines. The first part was implemented in line with Huang's improved algorithm for calculating the ZSIs, accounting for some patterns of interference that were missed in the original, among other improvements.

The run-time support consisted of some minor changes to the scheduler in order to support the different execution modes and also the deactivation of large parts of the SmartSAR timing protection which conflicted with the ZSS algorithm. The ZSI was implemented as an alarm triggering a high priority high criticality thread to set the system criticality mode. Perhaps this could have been implemented more efficiently by incorporating it further into the SmartSAR system, however, this was deemed to require too large changes to the existing systems.

The preparatory calculations for the ZSS systems were notably slower than the ones for AMC but still a lot faster than the EDF-DB precalculations. A lot of time was also spent on finding schedulable systems due to the lower schedulability of ZSS.

Simulation Support Systems

The main support system for the load simulation implemented during the project is the Code Generator, a Java programme that generates a task system with a specified number of tasks, utilisation, execution time, etc. It facilitates the automated creation of task systems and the application of the different scheduling algorithms on them, although there is no guarantee that a given task system once generated will be schedulable in any or all of the algorithms. The offline computations for all the algorithms were run within this Code Generator programme as part of the setup process.

In order to achieve the specified utilisation the UniFast algorithm outlined in [Bini and Buttazzo, 2005] has been employed, generating uniformly distributed task utilisations. The periods where created using the standard uniform distribution implemented in Java's Random class. After the parameters have been set and calculated respectively, the system is printed as C89 source code for a SmartSAR application.

4.3 Overhead Measurements

As previously mentioned, the overhead was measured using a logic analyser. The execution time of the three scheduling functions `Find_Next_Task`, `Insert_Ready_Task` and `Remove_Ready_Task` was measured by each of them sending a signal while executing. These signals were caught and recorded by the logic analyser. The combined time of the signals for each algorithm represents the time spent scheduling, i.e., the overhead.

As the system keeps track of each task, all the ready to run tasks are placed in lists. Depending on the scheduling algorithm, these lists were implemented differently.

Initially, the goal was to produce task systems using eight different numbers of tasks, (5, 10, 15, 20, 25, 30, 35, 40), set at 60% utilisation to ensure schedulability for ZSS, however task numbers greater than 30 also proved unschedulable in ZSS, as mentioned previously. Thus task systems with up to 40 tasks were generated for the other algorithms, but ZSS was limited to a maximum 30 tasks per system. There were ten systems generated for each algorithm at each number of tasks, thus there were 80 unique systems to run for AMC and EDF-DB each and 60 for ZSS.

Figures 4.1, 4.2 and 4.3 are all captured from the first 0.5 seconds of the systems runtimes. This is the worst case scenario, when all tasks are inserted into the ready list at once.

The single most used function is `Find_Next_Task` which is run every time the system is ready to continue after an interrupt. As seen in the graphs, EDF-DB is the fastest thanks to the linked list. This is shown in Figure 4.1.

When inserting and removing tasks from the ready queue, however, EDF-DB decreases in speed as the number of tasks increase, as can be seen in Figure 4.2. AMC and ZSS have basically the same overhead since they share the same scheduling at the base with different approaches for multiple criticality and preparatory work. Figure 4.3 shows a sum of the overhead of all three functions for each algorithm.

Figures 4.4, 4.6, 4.5 and 4.7 are based on data gathered over a period of eight seconds, excluding the initial worst case phase of execution. These measurements are thus closer to a more general execution situation. Here the generated systems differ. They were generated at 70% utilisation for AMC and EDF-DB while still at 60% for ZSS. Also, there were only three different amount of tasks, (10, 20, 30), and ten systems for each amount of tasks for each algorithm. The first three figures, 4.4, 4.6, and 4.5, show the individual costs of the insert, find next, and remove functions for each algorithm in turn. There is no great difference in these measurements compared to those performed during the shorter interval, save for the insert ready task function of the EDF-DB algorithm. An analysis of why this is the case can be found in Chapter 5.

Figure 4.7 puts the total overheads of all the algorithms in comparison to each other. The starkest impression is once again the low overhead of the EDF-DB algorithm which was much greater than both ZSS and AMC in the short period measurements but is now instead much lower. A slight difference can also be observed between AMC and ZSS to the advantage of ZSS, but as our measurements do not include overhead from the handling of the Zero Slack Instant it is doubtful whether the latter actually is faster. This was omitted due to limitations in the amount of time and resources available to spend on remodeling SmartSAR; the produced implementation proved too ponderous to yield a good comparative measurement and it was decided to exclude it from the measurements. It could also be an artifact of the difference in utilisation between the two systems, a result of ZSS's low schedulability.

4.4 Response Time Measurements

The response time was measured in the scheduling systems of SmartSAR, by recording the arrival and termination dates of the tasks, which was then printed to the target board console.

The total response times of ten systems schedulable for all three algorithms were measured, in order to see if there was any significant difference between them. These systems were generated at 50% utilisation with 30 tasks each. This resulted in the data described in Table 4.1. Postulating that the total response time can be modeled by a normal distribution, the mean (μ), standard deviation (σ) and their respective 95% confidence intervals can be found in Table 4.2.

System No.	ZSS	EDF-DB	AMC
1	56863	62094	58772
2	43501	47087	50369
3	47895	55770	79377
4	58517	67499	58180
5	53730	53991	67873
6	54440	61323	63870
7	46364	57622	55536
8	49073	50975	56772
9	48862	55065	52494
10	58640	67913	48742

Table 4.1 Response Time Measurements

Although there appears to be a slight difference, the 95% confidence intervals for the measurements overlap making it difficult to draw conclusions from them.

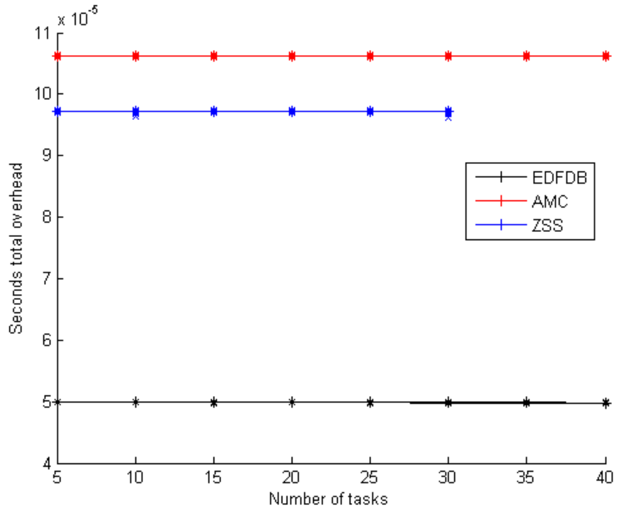

Figure 4.1 Time spent in the `Find_Next_Task function` for each algorithm

	ZSS	EDF-DB	AMC
μ	51789	57934	59199
σ	5354	6787	9171
μ_{ci}	47959, 55618	53079, 62789	52638, 65759
σ_{ci}	3683, 9774	4668, 12389	6308, 16743

Table 4.2 Response Time Estimates

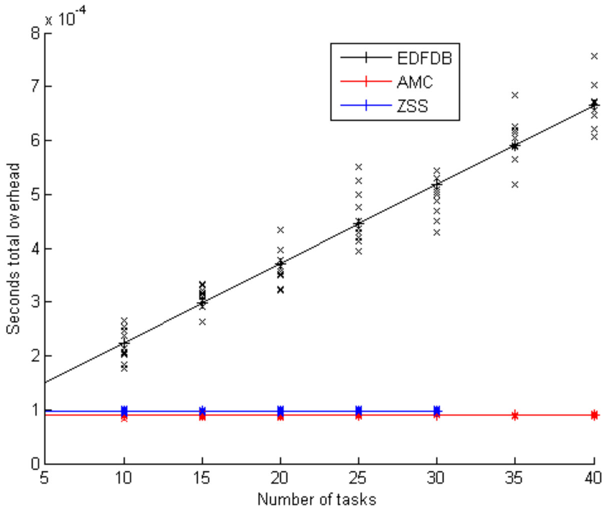

Figure 4.2 Time spent in `Insert_Ready_Task` and `Remove_Ready_Task` combined for each algorithm

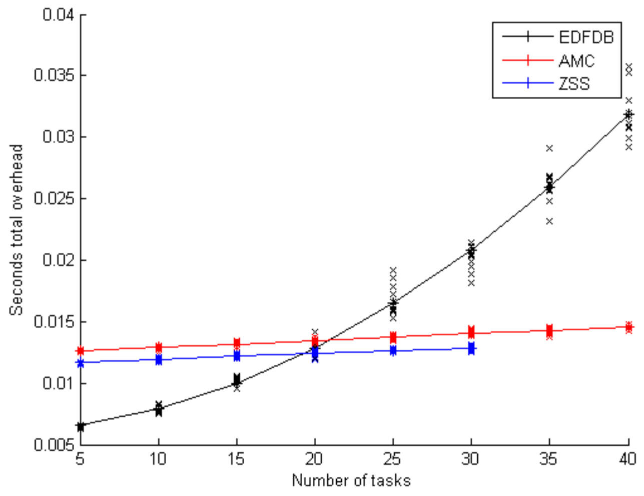

Figure 4.3 Overall overhead for each algorithm during the short measurement interval

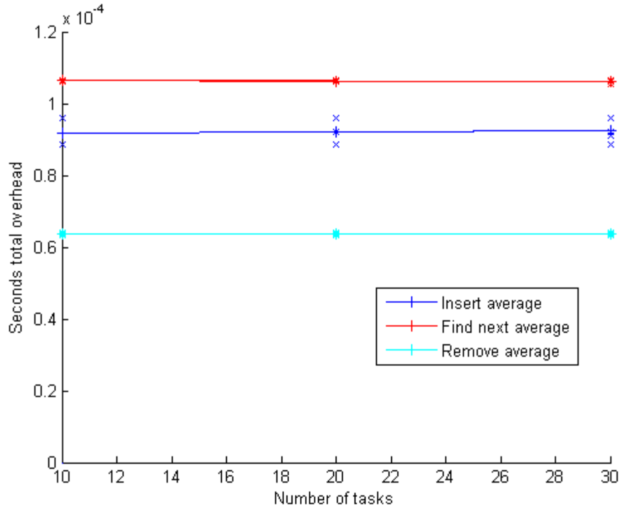

Figure 4.4 The three scheduling functions overhead - Long overhead run in AMC

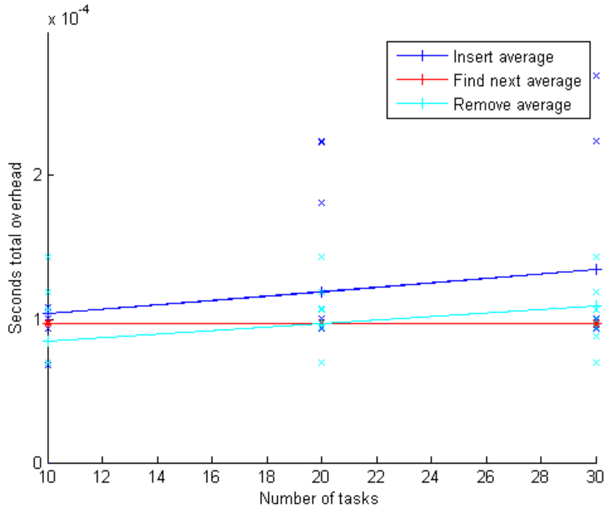

Figure 4.5 The three scheduling functions overhead - Long overhead run in ZSS

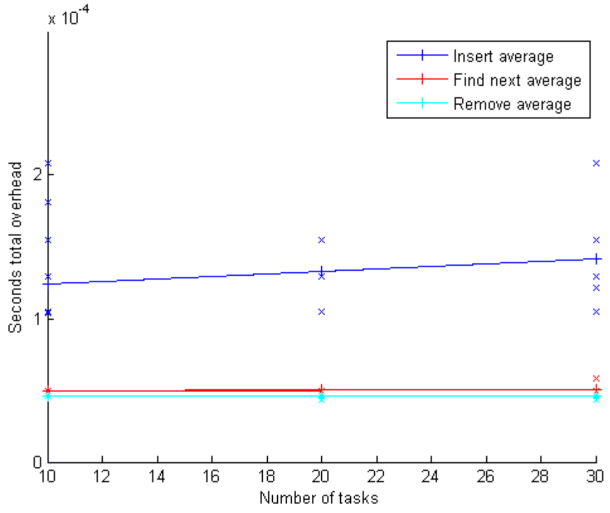

Figure 4.6 The three scheduling functions overhead - Long overhead run in EDF-DB

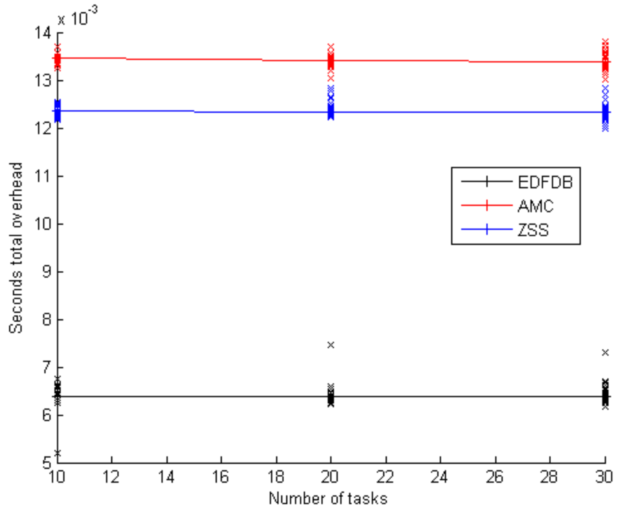

Figure 4.7 Overall overhead for each algorithm during the long measurement interval

5

Analysis

5.1 Measurement Data

It was discovered that during the running of AMC and EDF-DB systems, the ZSS system generation was unable to produce schedulable systems at higher than 60% utilisation and with more than 30 tasks each. The long preparatory calculations of EDF-DB and the time consuming running of both systems to achieve measurement data prevented any adjustments. The limitations of the ZSS scheduling system were not readily apparent before this moment.

During the task response time measurements, however, care was taken to assure good schedulability for all three algorithms.

5.2 Preparatory Calculations

The time consumption of the three algorithms varied a lot when doing the preparatory calculations. These calculations include changing priorities, adjusting deadlines and finding the ZSI among other things. When run on average, multi-core laptop computers the time consumed varied greatly, depending on which scheduling algorithm was chosen. The by far fastest was AMC due to the low iteration count of Audsley's algorithm. ZSS did take notably longer time. Most time consumed by far was EDF-DB. These results coincide with the iteration count estimates of the individual algorithms.

As noted earlier, the systems ran for twenty seconds each. Longer runtime would increase the preparatory calculations by a large amount of time, at least for EDF-DB and somewhat ZSS as well.

5.3 Overhead

Overall, it ought to be mentioned that the measurements discussed in this report are due to their intrinsic nature implementation specific; that is, the overhead generated by the system is of course highly dependent on how this system is implemented. This limits the applicability of the findings of this report but as the aim has been to use solutions perceived as commonplace for this manner of problems and situations, it should still provide some general guidance as to the advantages and disadvatages of the respective algorithms from an overhead perspective.

The most interesting feature of the short measurements, i.e. Figure 4.3 is the immense cost of the EDF-DB insert/remove functions for large amounts of tasks, especially compared to the relative cheapness of its find next function; the origins of this phenomenon will be discussed in the next paragraph. Even so it is still cheaper than the other two algorithms for task systems smaller than twenty tasks. ZSS has slightly less overhead than AMC in this worst-case comparison, however, its schedulability degrades rapidly, becoming almost zero for systems of more that 30 tasks with a utilisation of 60%, as proven by Huang in [Huang, 2012]. Nevertheless, it should be noted that the difference between ZSS and AMC in this initial phase is so small that it is perhaps more likely due to some minor details in the implementation than any large, overall difference in the algorithms.

Perhaps the most obvious difference when comparing the short and long measurement runs, i.e. Figure 4.2 compared to Figure 4.6, is the radical difference in the time consumption of the insert and remove functions of EDF-DB. This is quite natural considering that its implementation is reliant on a linked list – during the start up phase all tasks must be inserted into the list, which becomes quite onerous if they are numerous. For the other two algorithms there appears to be no significant difference when compared to the worst case overall costs, compare figures 4.3 and 4.7. The differences among the find next costs, as seen in comparing Figure 4.1 to Figures 4.4, 4.5 and 4.6, is once again negligible.

As opposed to the overall overhead measurements for the short interval, the differences among the measurements during the longer interval is much more pronounced, with the interesting effect that EDF-DB is in fact the cheapest from that perspective. This is most likely due to the fact that the previously mentioned worst case for the linked list dominates the earlier measurements, and the fact that the cheapness of the much more commonly executed find next function compensates for the relative expense of the insert function; this works especially well in the latter part of the programme as there are very few calls to the insert function at this stage of execution.

It should also be noted, that these comparisons rely on average costs per execution of the function, i.e. the total costs of the system are reliant on the individual configuration of the tasks, and how often they trigger a certain kind of operation.

These measurements paint EDF-DB as the most economical algorithm for most systems in the short and long run, although for certain systems, running perhaps for a very short time, it might be better to employ one of the other algorithms.

5.4 Task Response Time

From the total system response times of Table 4.1 it emerges. interestingly enough and perhaps somewhat counter intuitively, that the algorithm with the shortest total response time is in fact ZSS, the algorithm with the worst schedualbility. It is possible that the limited schedulability is tied to the shorter response times, the system being scheduled in a more compact manner. However, comparing the data for AMC and EDF-DB indicates this is not the case; EDF-DB has a higher schedulability than AMC but even so the results in Table 4.2 indicate that EDF-DB might have a shorter general response time. It should be noted, however, that under the postulate that the response time adheres to a normal distribution the difference between the algorithms is within the 95% confidence interval, as demonstrated in table 4.2. It is thus quite risky to draw any conclusions from this particular data, and any definite conclusions are perhaps best deferred until further tests have been performed with a bigger data set.

6

Conclusion

Having examined three of the elicited algorithms – Zero-Slack Scheduling, Adaptive Mixed Criticaliy and Earliest Deadline First Demand Bound – from the overhead and response time measurement perspectives we are left with somewhat mixed results. From an overhead point of view, both AMC and EDF-DB are interesting alternatives. EDF-DB is quicker in run-time but the precalculations, being contingent on the hyperperiod or the entire running duration of the system, might prove crippling for large systems. AMC's precalculations are quicker, but the overhead is also somewhat more expensive for the general case. ZSS manages to combine time consuming precalculations with a low schedulability and an overhead which is most likely more expensive than AMC. It does appear to have a shorter overall response time, but these measurements do not, unfortunately, stand up to statistical scrutiny as the confidence intervals for the different algorithms overlap, rendering them inconclusive.

6.1 Future Work

Perhaps the most obvious area for future work is the response time measurements. Furher data could perhaps establish with a higher degree of confidence that there is a difference , or at least that there is not. Should it be the case that there is in fact a differece, and that ZSS is slightly cheaper as these results seem to hint at, it might be interesting to investigate the root of this phenomenon.

It has also emerged during the project that some of the algorithms appear, at cursory examination, to be quite sceptical in their estimation of the interference provided by the tasks; especially if the tasks are considered to have a variable execution time. It should be possible to estimate this scepticism through empirical measurements over the whole hyperiod, i.e. running systems deemed by the algorithms to be unschedulable and checking whether they actually fail. This accuracy of estimation is an important factor pertaining to the usefulness of the algorithms that has been left outside the scope of this project due to the unwieldiness of the hyperperiod, but might be interesting to study in order to complete the picture.

Bibliography

Adrian North, L. M. (2001). “Psychologists’ trials find music tempo affects productivity”. *http://www.le.ac.uk/press/press/moosicstudy.html*.

Årzén, K.-E. (2011). *Real Time Control Systems*. Department of Control, Lund University, Lund, Sweden.

Baruah, S., V. Bonifaci, G. D’Angelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie (2012). “The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems”. In: *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pp. 145 –154. DOI: `10.1109/ECRTS.2012.42`.

Baruah, S. K., A. K. Mok, and L. E. Rosier (1990). “Preemptively scheduling hard-real-time sporadic tasks on one processor”. In: *Real-Time Systems Symposium, 1990. Proceedings., 11th*. IEEE, pp. 182–190.

Baruah, S., A. Burns, and R. Davis (2011). “Response-time analysis for mixed criticality systems”. In: *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pp. 34 –43. DOI: `10.1109/RTSS.2011.12`.

Bini, E. and G. C. Buttazzo (2005). “Measuring the performance of schedulability tests”. *Springer Science + Business Media, Inc.* **30**, pp. 129 –154.

Burns, A. and R. Davis (2013). “Mixed criticality systems: a review”. *Department of Computer Science, University of York, Tech. Rep.*

Chattopadhyay, B. “Preemptive uniprocessor scheduling for mixed-criticality systems”.

Ekberg, P. and W. Yi (2012). “Measuring the performance of schedulability tests”. In: *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pp. 135–144. DOI: `10.1109/ECRTS.2012.24`.

Ekberg, P. and W. Yi (2014). *Springer Real-Time Systems* **30**:50, pp. 48 –86.

Extreme Programming Pocket Guide (2003). Pocket References Series. O’Reilly Media. ISBN: 9780596004859. URL: `http://books.google.se/books?id=WtOFlVWrEXkC`.

Guan, N., P. Ekberg, M. Stigge, and W. Yi (2011). “Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems”. In: *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pp. 13 –23. DOI: `10.1109/RTSS.2011.10`.

Huang, H. (2012). *MCFlow: Middleware for Mixed-Criticality Distributed Real-Time Systems*. PhD thesis. WASHINGTON UNIVERSITY IN ST. LOUIS.

Li, H. and S. Baruah (2010). “An algorithm for scheduling certifiable mixed-criticality sporadic task systems”. In: *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pp. 183 –192. DOI: `10.1109/RTSS.2010.18`.

Li, H., P. Lu, M. Yao, and N. Li (2009). “Smartsar: a component-based hierarchy software platform for automotive electronics”. In: *Embedded Software and Systems, 2009. ICESS’09. International Conference on*. IEEE, pp. 164–170.

Niz, D. de, K. Lakshmanan, and R. Rajkumar (2009). “On the scheduling of mixed-criticality real-time task sets”. In: *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pp. 291 –300. DOI: `10.1109/RTSS.2009.46`.

Vestal, S. (2007). “Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance”. In: *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pp. 239 –243. DOI: `10.1109/RTSS.2007.47`.

Lund University **Department of Automatic Control** **Box 118** **SE-221 00 Lund Sweden**	*Document name* MASTER´S THESIS *Date of issue* September 2014 *Document Number* ISRN LUTFD2/TFRT--5959--SE
Author(s) Carl Cristian Arlock Edward Linderoth-Olson	*Supervisor* Anton Cervin, Dept. of Automatic Control, Lund University, Sweden Zonghua Gu, Zhejiang University, Kina Karl-Erik Årzén, Dept. of Automatic Control, Lund University, Sweden (examiner) *Sponsoring organization*

Title and subtitle

A Practical Comparison of Scheduling Algorithms for Mixed Criticality Embedded Systems

Abstract

With the consolidation of automotive control processes onto single high-performance ECUs the issue of running, and thus scheduling, processes of varying criticality on a single CPU has moved to the fore. This has resulted in a number of new algorithms for scheduling such systems, for example Adaptive Mixed Criticality (AMC). This project attempts to measure the performance of some of these algorithms on a single core embedded system CPU and compares them in order to shed some light on their different advantages and disadvantages.

Keywords
Unicore, Scheduling, Embedded systems, Mixed Criticality, AUTOSAR, HCS12X.

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title 0280-5316	*ISBN*

Language English	*Number of pages* 1-48	*Recipient's notes*
Security classification		

http://www.control.lth.se/publications/