



EUROPEAN COMMISSION

SEVENTH FRAMEWORK PROGRAMME

Theme: ICT

Small or medium-scale focused research projects (STREP)

FP7-ICT-2013-10

Objective ICT-2013.6.5 Co-operative mobility

a) Supervised Automated Driving

GA No. 612035

Interoperable GCDC AutoMation Experience

Deliverable No.	i-GAME D2.4	
Deliverable Title	Interoperable robustness measures for safety-integrity levels (SILs)	
Dissemination level	Public	
Written By	Tarun Gupta (TU/e)	01-03-2015
Checked by	Erik J. Luit (TU/e)	02-03-2015
	Reinder J. Bril (TU/e) (TU/e)	04-03-2015
	Cristofer Englund (Viktoria)	07-03-2015
	Alejandro I. Morales Medina (TU/e)	25-03-2015
Approved by	Almie van Asten (TNO)	01-04-2015
Status	Final	31-03-2015

Please refer to this document as:

DEL150331_i-GAME_D2.4 Interoperable robustness measures for SILs

Acknowledgment:

The author would like to thank Erik J. Luit (TU/e) and Reinder J. Bril (TU/e) for their guidance and valuable contributions to the work presented in this document and Martijn M.H.P. van den Heuvel (TU/e) for discussions. Moreover, the author would like to thank Cristofer Englund (Viktoria) and Alejandro I. Morales Medina (TU/e) for their comments on an earlier version of this document.

Disclaimer:

i-GAME is co-funded by the European Commission, DG Research and Innovation, in the 7th Framework Programme. The contents of this publication is the sole responsibility of the project partners involved in the present activity and do not necessarily represent the view of the European Commission and its services nor of any of the other consortium partners.

Executive Summary

The i-GAME solution towards automated driving is based on so-called supervisory control, that provides both event-driven control to initiate vehicle manoeuvres (e.g. a car want to merge on a highway) and real-time control to execute the manoeuvres (e.g. vehicles make space for the merging vehicle and the merging vehicle steers into the empty space). Supervisory control systems inherently support applications (or functionalities) of mixed criticality, e.g. different safety integrity levels (SILs); co-operative automated driving is inherently critical whereas informative drive assistance services are typically non-critical. Traditionally, safety-critical and non-critical applications are segregated in the architecture of vehicles, e.g. by using separate ECUs connected to different networks, which are on their turn connected via gateways. Virtualization of shared resources, i.e. giving each application the illusion that it has all resources exclusively available, is a major step towards providing isolation between applications. From a real-time perspective, virtualization is addressed by means of hierarchical scheduling frameworks (HSFs). The integration of mixed-criticality functionality brings new challenges beyond robustness of individual functionality as provided by HSFs, however. In particular, the cooperative automated in-vehicle platform shall also satisfy safety requirements.

In the ISO 26262 functional standard for automotive systems, published in November 2011, so-called ASILs (Automotive Safety Integrity Levels) are specified. Given the emerging need to certify safety-critical systems according to this standard, the real-time systems community has started to develop novel models and scheduling strategies. Incorporation of mixed-criticality in HSFs is still at its infancy, however. In this document, which is a deliverable of Task 2.3 "Safety integrity levels in hierarchical scheduling frameworks" of Work package 2 "Interoperable vehicle", describes the results of our initial efforts in incorporating mixed criticality in a cooperative automated in-vehicle platform, i.e. in a real-time operating system (RTOS) supporting a HSF. The document presents a review of the state-of-the art in the real-time systems literature on mixed-criticality scheduling schemes, and an exploration of the feasibility of extending an existing HSF-enabled RTOS with mixed-criticality through the development of a prototype system. This document starts by describing three non-functional requirements of the automotive domain that are directly related to both mixed criticality and virtualization (or HSFs), being certification, cost reduction, and runtime robustness. Next, it summarizes preliminary models and definitions for mixed-criticality scheduling schemes as have been described in the literature. Based on (i) identified system requirements and (ii) the specification of an extension of an existing mixed-criticality scheme satisfying these requirements, an in-depth description is presented of the design and implementation of a prototype system using Linux and a loadable kernel module.

Contents

1	Introduction	7
1.1	Project context: i-Game	7
1.2	Scope	8
1.3	Document Outline	8
2	Domain Analysis	9
2.1	MC - Mixed Criticality	10
2.2	Non-Functional Aspects of The Automotive Domain	11
2.2.1	Certification: ASIL	11
2.2.2	Cost Reduction	13
2.2.3	Runtime Robustness	14
2.3	Derived application drivers	17
3	Preliminary Models and Definitions	18
3.1	Generic Mixed Criticality Model	18
3.1.1	Vestal's Scheme - 2007	20
3.1.2	Baruah and Vestal - 2008	20
3.2	SMC Scheme - 2011	20
3.3	AMC Scheme - 2011	21
3.4	Relaxed-AMC Scheme - 2012	21
3.5	Practical MC Scheme - 2013	22
3.6	Overview	22
4	System Requirements	24
4.1	Requirements for the AMC scheme	24
4.2	Level-up trigger handler	25
4.3	Level-down trigger handler	25
4.4	Domain specific requirement	25
5	Scheme Semantics	26
5.1	AMC* Scheme	26
5.2	AMC* Scheme: Expected Behaviour	30
5.2.1	Task state diagram	30
5.2.2	Job state diagram	33
5.2.3	Level Handler state diagram	35
5.2.4	Sequence diagram representing state changes	37
5.3	Cases for the Level-up trigger	38
5.4	AMC* Scheme: Scenarios	38
5.4.1	Scenario 1: Case 1: Task Suspend (including Job Abort) and Task Resume of Ready task	40
5.4.2	Scenario 2: Case 2 and Case 1: Job Abort of Contending job	42

5.4.3	Scenario 3: Case 2: Job Abort of Pre-empted Job	44
5.4.4	Scenario 4: Case 2: Task Suspend of Waiting Task and Task Resume	46
5.4.5	Scenario 5: Case 2: Multiple criticality level change at once and Error Condition	48
5.4.6	Scenario 6: Case 1: Level-down trigger due to a Level-up trigger	50
6	ExSched Framework	52
6.1	ExSched Module	52
6.2	ExSched Library	53
6.3	Hierarchical Scheduling Framework	54
6.4	ExSched: Plug-in development	55
6.5	Extending the support for plug-in development	58
6.5.1	Run-time Monitoring	58
6.5.2	Task Management Services	59
6.5.3	Extended Plug-in Interface	60
7	System Design	63
7.1	Design Influencers	63
7.2	AMC* Plugin Module structure	63
7.3	State Diagrams	66
7.3.1	Task State Diagram	67
7.3.2	Job State Diagram	70
7.3.3	Level Handler State Diagram	73
7.4	Sequence Diagrams	75
7.5	Additional design decisions	76
7.5.1	Alternative: Killing and Restarting a Task	77
7.6	Process view	77
7.6.1	Timers and Interrupts	77
7.6.2	Race Conditions	77
7.7	Scenarios	78
8	Implementation	80
8.1	Linux Scheduler and Kernel API	81
8.2	Generic Functionality	82
8.2.1	Runtime Monitoring	82
8.2.2	Detecting Idle Time	82
8.2.3	Task Suspend-Resume	85
8.2.4	Job Abort	88
8.3	AMC* Plugin Module	93
8.3.1	Storing Task Information	93
8.3.2	Race Conditions	99
9	Conclusions	100
9.1	Results and conclusions	100
9.2	Future work	100
	Appendices	105
A	Hierarchical Scheduling Framework	106
B	Implementation Set-up	108
B.1	Operating System Set-up	108
B.2	Extended ExSched Set-up	109
B.3	Grasp Tool Set-up	110

List of Tables

1.1	i-Game Individual Participants	7
3.1	Comparison of Mixed Criticality (MC) schemes	23
5.1	Overview of scenarios	39
5.2	AMC*: Task set 1	41
5.3	AMC*: Task set 2	43
5.4	AMC*: Task set 3	45
5.5	AMC*: Task set 4	47
5.6	AMC*: Task set 5	49
5.7	AMC*: Task set 6	51
6.1	Methods of the ExSched Module which allocate the CPU to the highest priority job selected by the scheduler for execution.	53
6.2	ExSched-API: ExSched Library functions are referred to by a different name in the ExSched Module.	54
6.3	Method Calls exposed to the ExSched Plug-ins.	57
6.4	API exposed by the ExSched Module to install and un-install a plug-in.	57
6.5	Run-time Monitoring. These three methods are local to the ExSched Module and must therefore be implemented in the ExSched Module.	59
6.6	API exposing Task Management services to plug-ins. These are functions that must be exposed by the ExSched Module and are declared with the "extern" keyword.	60
6.7	Extension to the API for ExSched Plug-ins.	61
6.8	Updated API exposed by the ExSched Module to install and un-install a plug-in.	62
7.1	AMC* Library.	65
7.2	Extended API exposing Task Management services to plug-ins.	65
7.3	Callback functions implemented in the AMC* Plugin Module.	65
8.1	Kernel-API for ExSched Module.	82
8.2	Methods of the ExSched Module which allocate or deallocate the CPU to a job.	85
8.3	Variables and methods introduced in the Task Skeleton.	92
8.4	API to disable and enable the interrupts.	99
A.1	HSF Comparison	106

List of Figures

1.1	i-Game participant organization chart	8
2.1	Key Drivers	11
2.2	Basic notations concerning timing analysis of systems [34]	13
2.3	Example to illustrate the impact of Certification and Cost reduction on analysis and schedulability. Quote from [13]	14
2.4	Key Drivers revisited	17
5.1	Task state diagram.	31
5.2	A job state diagram of task τ_i	33
5.3	A state diagram of the Level Handler. NQ is the number of jobs in the ready queue.	35
5.4	Sequence diagram representing the triggers and actions causing the state changes in the three state diagrams.	37
5.5	AMC* Scheme: Criticality level-up change to L^2 at time 25ms due to a job of Task 1 of representative criticality level L^1 and a level-down change due to a level- L^2 idle time at 52ms.	41
5.6	AMC* Scheme : Criticality Level-up change to L^2 at time 57ms due to a job of Task 2 and then at time 61ms to L^3 due to a job of Task 2 of representative criticality level L^2 and a level-down change due to a level- L^3 idle time at time 73ms.	43
5.7	AMC* Scheme : Criticality level-up change to L^2 at time 23ms due to a job of Task 1 of representative criticality level L^2 and a level-down change due to level- L^3 idle time at time 25ms.	45
5.8	AMC* Scheme : Criticality Level-up change to L^2 at time 15ms due to a job of a task of representative criticality level L^2 and a level-down change due to a level- L^2 idle time at time 48ms.	47
5.9	AMC* Scheme : First a criticality level-up (multiple) change to L^3 at time 10ms due to a job of a task of representative criticality level L^3 and a level-down change due to a level- L^3 idle time at time 16ms. Then an error condition due to job of a task of representative criticality level L^3 at time 36ms.	49
5.10	AMC* Scheme: Criticality level-up change to L^3 due to a job of a task of representative criticality level L^2 at time 47ms and a level-down change due to a level- L^3 idle time at time 47ms.	51
6.1	ExSched: Structural components [8]	52
6.2	ExSched: sequence diagram representing the invocation of the callback functions.	56
7.1	Static structure extended with the AMC* Plugin Module.	64
7.2	ExSched: task state diagram.	68
7.3	Job state diagram.	71
7.4	Level Handler state diagram. We have used the Latex [4] notations here.	73
7.5	Sequence diagram of a criticality level-up change representing the calls which trigger state changes in the three state diagrams.	75

7.6	Sequence diagram of a criticality level-down change representing the calls which trigger state changes in the three state diagrams.	76
7.7	AMC* Scheme : Criticality Level-up change to L^2 and then to L^3 due to a task of representative criticality level L^2 and a level-down change due to a level- L^3 idle time	79
7.8	AMC* Scheme : Execution time of the job abort mechanism. In our set-up the execution time of job abort of Task 3 is less than one microsecond.	79
8.1	Sequence diagram representing the functionality of Runtime Monitoring and detecting Idle Time.	84
8.2	User space ExSched Task	89
8.3	AMC* Task	91
8.4	Sequence diagram for level-up change functionality.	96
8.5	Sequence diagram for level-down change functionality.	98

Acronyms

AMC Adaptive Mixed Criticality. 13, 24–26, 29

ASIL Automotive Safety Integrity Level. 14, 15

ASIL A Automotive Safety Integrity Level A. 14, 15

ASIL D Automotive Safety Integrity Level D. 14, 15

BCET Best-Case Execution Time. 16

CA Certification Authorities. 14–16

CAPA Criticality As Priority Assignment. 12

CPU Central Processing Unit. 12, 16, 18, 19, 61

D&C Dynamics and Control. 8

ECU Electronic Control Units. 11, 16

EDF Earliest Deadline First. 55, 56

FPPS Fixed-Priority Pre-emptive Scheduling. 55, 56, 92

HS Hierarchical Scheduling. 19

HSF Hierarchical Scheduling Framework. 19

MC Mixed Criticality. 5, 11, 13, 14, 16–18, 20, 21, 23–27

MCS Mixed Criticality System. 16, 17, 19, 21, 25, 66

PDEng Professional Doctorate in Engineering, Software Technology. 8

QM Quality Management. 15

RESCH REal-time SCheduler. 55

SAN System Architecture and Networking. 8

SMC Static Mixed Criticality. 13, 23–26

TU/e Eindhoven University of Technology. 8

WCET Worst-Case Execution Time. 12, 13, 16, 23–30, 32, 44, 46, 48, 50, 52, 54, 61

Chapter 1

Introduction

This report provides a comprehensive overview of, and explains the research work done for, the Professional Doctorate in Engineering, Software Technology (PDEng) assignment within the i-Game [27] project. This chapter provides an introduction to the project context, a high-level description of the goals of i-Game and a brief outline of the document.

1.1 Project context: i-Game

i-Game aims to speed up real-life implementation of interoperable cooperative automated driving, enabled by wireless communication. The objective of i-Game project, in general, is to develop generic, fault-tolerant, and resilient technologies for automated driving and supporting cooperative applications, focusing on the supervisory control level.

The individual participants of the i-Game project are listed in Table 1.1.

Table 1.1: i-Game Individual Participants

Abbreviation	Full name
TNO	Nederlandse Organisatie voor toegepast-natuurwetenschappelijk onderzoek
Eindhoven University of Technology (TU/e)	Technische Universiteit Eindhoven
Viktoria	Viktoria Swedish ICT
IDIADA	Applus+ IDIADA

There are two departments within the TU/e that are involved in the i-Game project: the Department of Mechanical Engineering (WTB - Werktuigbouwkunde) and the Department of Mathematics and Computer Science (WIN - Wiskunde & Informatica). The research groups within the departments involved are Dynamics and Control (D&C) and System Architecture and Networking (SAN) respectively. The PDEng assignment is executed within the SAN group of the Department of Mathematics and Computer Science. The assignment falls in Work Package 2 (WP2) of i-Game's project plan. The relevance of the PDEng assignment particularly falls under the task of "Safety integrity levels in hierarchical scheduling frameworks" i.e. T2.3 of WP2.

Within i-Game, the cooperative automated vehicle system is meant to contain the basic technology needed for cooperative automated driving and standard interfaces to informative services. Given its role, the system supports applications of mixed criticality, e.g. different safety levels; automated driving is critical whereas informative services are typically non-critical. To satisfy the safety requirements, the cooperative automated in-vehicle platform shall, therefore, not only provide basic mechanisms for robustness, such as a Hierarchical Scheduling Framework (HSF), but also mechanisms for mixed criticality.

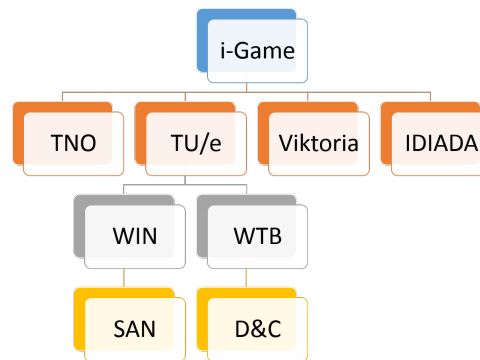


Figure 1.1: i-Game participant organization chart

1.2 Scope

The scope of this document is to present Mixed Criticality (MC) scheduling schemes discussed in literature and to select one of these schemes. We briefly cover the issues related to MC. In the scope of i-Game we focus on addressing the aspects of MC by first selecting and then extending a real-time scheduler. The intention is to realize the mechanisms for Criticality-Level change in this extended scheduler. The document describes how the criticality-level change mechanisms have been conceptualized and realized.

1.3 Document Outline

Chapter 2 explains concepts and technologies that are relevant for this project.

Chapter 3 presents the preliminary models and various Mixed Criticality schemes discussed in the literature.

Chapter 4 documents the functional and non-functional requirements of this project.

Chapter 5 documents the behaviour of the chosen MC scheme namely the AMC* scheme.

Chapter 6 describes layers and components of the existing scheduling system and the identified extensions.

Chapter 7 The system design and design decisions are presented here.

Chapter 8 documents the realization of the design. The implementation of the AMC* scheme is described here. Class and sequence diagrams are used to illustrate the design.

Chapter 9 documents the conclusions and the future work.

Chapter 2

Domain Analysis

A growing trend in the automotive domain is a feature intensive vehicle. These features may be safety related, driver assistance related, connected services or multimedia and entertainment related. With the constant increase in the size of the feature list of a vehicle, computation also needs to grow. A typical car is controlled by over 100 million lines of code executing on close to 100 Electronic Control Units (ECU) [22]. It is not desirable to add more ECUs in the vehicle to provide computation power mainly because of the following four reasons:

- **Weight constraints:** adding more hardware increases the weight of the vehicle thereby degrading the performance of the vehicle, which is not desirable. There is a limit to cabling that can be added to the vehicle. Increasing length and number of cables contribute to the weight of the vehicle, which is the most important constraint factor.
- **Integration issues:** by adding more ECUs complexity is added to the system. This leads to issues related to system integration from a temporal viewpoint.
- **Cost reduction:** reducing the cost of the system is also important in the context of the automotive domain, so integrating functionality of multiple ECUs into one bigger (in terms of computing power) ECU (single or multi-core) is desirable.
- **Number of Points of Failure:** adding more ECUs means new connections in the system leading to an increased number of Points of Failure, due to the number of connectors. An increased network load also contributes to the computation needs.

This chapter presents some of the supporting technological developments for software and hardware optimizations.

It is evident that various types of applications run in an automotive system. Applications related to driver assistance such as monitoring the state of vehicle components and the vehicle's surroundings, navigation and traffic related services, temperature control and cruise control. Applications running within a contemporary vehicle are inherently complex due to enormous amounts of communication and synchronization required between the vehicle subsystems. A next generation vehicle, like an autonomous self-driving car, takes the complexity of applications to the next level.

The standard applications running in a vehicle strongly demand a strict time-bound response. Different applications run within a vehicle with different temporal requirements. Some applications have strict deadlines and some do not. In the Section 2.1, MC is introduced and in the following section (Section 2.2) some non-functional drivers of MC, with respect to automotive domain, are highlighted. The relevance of the key-drivers is discussed in Section 2.2.1, Section 2.2.2 and Section 2.2.3.

2.1 MC - Mixed Criticality

Criticality is a designation of the level of assurance against failure needed for a system component. A mixed criticality system (MCS) is one that has two or more distinct levels of criticality (for example safety critical, mission critical and non-critical)[16].

Safety Integrity Level (SIL) and Automotive Safety Integrity Level (ASIL) (an adapted version of SIL) are classification schemes in which levels are identified.

In this report we assume that each subsystem is typically associated to one application. Each application may release one or more tasks. A task is associated to an individual component of a subsystem. Not all tasks of an application are of the same criticality level.

Tasks may have different criticality, e.g., safety related tasks will, in general, have higher criticality than navigation related tasks. Therefore, it is more important to adhere to the deadlines of safety critical tasks than those of others. An example of a criticality classification is:

- **Safety Critical:** When a task is not finished within its deadline, a potential threat to the life of the passengers may arise. E.g., control of the air bags.
- **Mission Critical:** When a task is not finished within its deadline, a potential threat to the safe operation of the system may arise or may causes a system damage leading to an incomplete mission. E.g., Parking assistance(sensors), Navigation and traffic services.
- **Non-Critical:** When a task is not finished within its deadline and does not pose any life or mission threatening situations, such a task is classified as non-critical. They cause user dissatisfaction at worst E.g., Entertainment, Multimedia.

In the event of a task exceeding its Worst-Case Execution Time (WCET), the schedulability can no longer be guaranteed. In such an event, it is advisable to adhere to the deadlines of safety-critical tasks potentially at the cost of first non critical and eventually mission critical tasks as well. To detect that a task exceeds its WCET requires monitoring support. In Section 2.2.3, we introduce runtime monitoring.

The WCET of a task cannot be precisely determined. Various methods for close approximation of the WCET have been proposed in literature that all produce a conservative estimate of the WCET. A common assumption that tasks would never exceed their estimated execution times does not hold true in practice. Tasks may occasionally exceed their execution time estimates and cause system overload situations. One way to overcome overload situations is resource reservation, e.g., reserving Central Processing Unit (CPU) time. The enforcement of resource reservation protects high-criticality tasks from faults of low-criticality tasks. However, resource reservation may also cause a reverse effect, i.e. reserving time for lower criticality tasks in situations where high criticality task require more resources than estimated. This reverse effect leads to the problem commonly referred to in literature as *Criticality Inversion*.

A traditional solution is to use a Criticality As Priority Assignment (CAPA) scheme as a means to overcome the problem of criticality inversion. However, this priority ordering may not be optimal leading to a poor CPU utilization.

In 2007 Vestal [33] presented pre-emptive scheduling of a Mixed/Multi Criticality System(MCS). Vestal proposed to use the technique of period transformation to provide assurances to higher criticality tasks in case their priority ordering is not criticality driven, i.e. when the safety critical tasks are running at lower priorities than some less critical tasks. Vestal proved that a modified version of Audsley's priority assignment algorithm [9] is appropriate for such systems. It was evident from Vestal's work that a mechanism (discussed in detail in Section 2.2.3) to detect a task exceeding its WCET is essential to realize an MCS. Once a task exceeding its allowed WCET is detected, higher assurances must be provided to higher criticality tasks. This is achieved by changing the criticality-level of the system to a higher criticality level. Changing the system's criticality to a higher level is called *criticality level-up change* and to a lower level is called *criticality level-down change*. A criticality level change consists of the following three things:

1. Detecting the need for a criticality-level change
2. Handling the lower criticality tasks

3. Handling higher criticality tasks

Various schemes for scheduling MCS have been proposed after that. Notable ones are Static Mixed Criticality (SMC) [11] and Adaptive Mixed Criticality (AMC) [13]. In [13], Baruah *et al.* prove that the AMC scheme is better than SMC in terms of schedulability. They also prove that a version of Audsley's priority assignment algorithm is the most appropriate one for the SMC and AMC schemes.

In [31] Santy *et al.* improved the AMC scheme by relaxing some of the assumptions of the AMC scheme and proposed the method of Latest Completion Time (LCT). This method ensured fewer job drops in case of a criticality level-up change. They also prove that an idle period will be an appropriate moment to perform a criticality level-down change to the lowest criticality level.

Graydon *et al.* [21] have made an attempt in bringing a structure to the research being done in MC context. They have proposed common understanding and definitions in the MC domain.

To summarize, in a mixed criticality system it is important to define the mechanisms required for detecting temporal faults, especially tasks exceeding their allowed WCET. Upon detection, some mechanisms for runtime support are required in order to provide higher assurances corresponding to the criticality levels/classification of the tasks. In Chapter 3 we will discuss various existing models and their key characteristics in detail.

2.2 Non-Functional Aspects of The Automotive Domain

Certification, cost reduction and runtime robustness are three non-functional aspects of the automotive domain that are directly related to MC and are called key drivers for MC (Figure 2.1). The following subsections (i) describe the key drivers, (ii) describe the problems related to these key drivers and (iii) how MC helps to achieve the goals associated with the key drivers.



Figure 2.1: Key Drivers

2.2.1 Certification: ASIL

Certification of safety-critical systems is an important driver for the timing behaviour, especially worst-case behaviour (during run-time), of tasks in execution. Certification is also important since software designers and engineers have to be mindful of the requirements of Certification Authorities (CA), in order to determine schedulability of tasks. To quote from [1]:

Automotive Safety Integrity Level (ASIL) refers to an abstract classification of inherent safety risk in an automotive system or elements of such a system. ASIL classifications are used within ISO 26262 to express the level of risk reduction required to prevent a specific hazard, with Automotive Safety Integrity Level D (ASIL D) representing the highest and Automotive Safety Integrity Level A (ASIL A) the lowest.

Severity Classifications (S):

- S0** No Injuries
- S1** Light to moderate injuries
- S2** Severe to life-threatening (survival probable) injuries
- S3** Life-threatening (survival uncertain) to fatal injuries

Risk Management recognizes that consideration of the severity of a possible injury is modified by how likely the injury is to happen; that is, for a given hazard, a hazardous event is considered a lower risk if it is less likely to happen. Within the hazard analysis and risk assessment process of this standard, the likelihood of an injurious hazard is further classified according to a combination of

exposure (E) (the relative expected frequency of the operational conditions in which the injury can possibly happen) and

control (C) (the relative likelihood that the driver can act to prevent the injury).

Exposure Classifications (E):

- E0** Incredibly unlikely
- E1** Very low probability (injury could happen only in rare operating conditions)
- E2** Low probability
- E3** Medium probability
- E4** High probability (injury could happen under most operating conditions)

Controllability Classifications (C):

- C0** Controllable in general
- C1** Simply controllable
- C2** Normally controllable (most drivers could act to prevent injury)
- C3** Difficult to control or uncontrollable

ASIL can be expressed as:

$$ASIL = Severity \times (Exposure \times Controllability)$$

In terms of these classifications, an ASIL D hazardous event is defined as an event having reasonable possibility of causing a life-threatening (survival uncertain) or fatal injury, with the injury being physically possible in most operating conditions, and with little chance the driver can do something to prevent the injury. That is, ASIL D is the combination of S3, E4, and C3 classifications. For each single reduction in any one classification from its maximum value (excluding reduction of C1 to C0), there is a single level reduction in the ASIL from D. [For example, a hypothetical uncontrollable (C3) fatal injury (S3) hazard could be classified as ASIL A if the hazard has a very low probability (E1).] The ASIL level below A is the lowest level, Quality Management (QM). QM refers to the standard's consideration that below ASIL A, there is no safety relevance and only standard Quality Management processes are required.

Both a system designer and an engineer tend towards measured or slightly higher execution time estimates for all tasks (safety-critical, mission-critical and non-critical). Whereas, during the certification process, a CA requires more pessimistic execution time estimates for safety-critical and mission-critical tasks than for non-critical tasks. This difference in expectation gives rise to problems. Using Figure 2.2 presented by Wilhelm *et al.* [34] the difference in expectations of an Engineer and a CA can be better explained. The figure depicts the variation in the measured, actual and bounded execution time of a task.

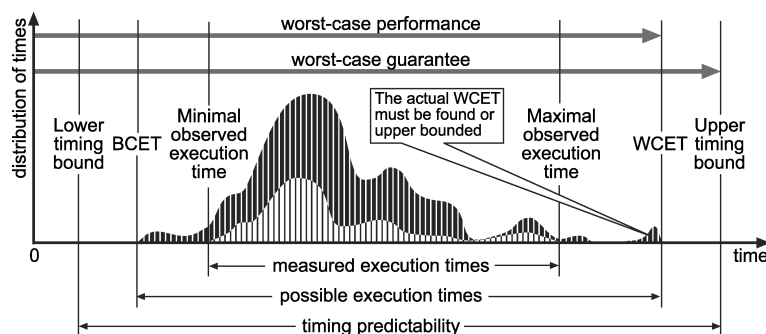


Figure 2.2: Basic notations concerning timing analysis of systems [34]

A common method to estimate execution time of a task is to measure the execution time for a subset of possible executions. This approach gives the measured/observed execution times, which are not safe for hard real-time tasks.

Other methods use detailed measurements of different parts of the task to give better estimates of its Best-Case Execution Time (BCET) and its WCET.

To make the timing analysis feasible and to compute bounds on the execution time of a task, an abstraction of this task can be used. The amount of information lost during the abstraction process depends on the overall system properties and method used for timing analysis. The WCET bound achieved by task abstraction represents the worst-case guarantee.

The CAs are concerned with the functional and temporal correctness of safety-critical and mission-critical tasks of the system, whereas the designer wishes to ensure that the non-critical tasks are also able to run. Hence, there is a twofold difference in expectations of a software designer and engineer vs the CA. (i) with respect to the execution times, i.e. measured/observed execution times by engineers and pessimistic upper bounds by CAs and (ii) based on criticality of tasks, since certification authorities do not care about non-critical tasks. Both of these factors lead to impact on schedulability of non-critical tasks.

The classical approach, to bridge this gap in expectations, is probably that tasks of different criticality are executed on separate ECUs. A question arises is whether this approach is cost effective. We shall revisit this question in the next subsection (Section 2.2.2).

MC scheduling was first presented by Vestal [33] in the context of the avionics domain. The DO-178B, published by RTCA, Incorporated, is a software development process standard, which assigns criticality levels to tasks based on their effect categorization. Vestal's analysis refers to the RTCA DO-178B standard. His analysis presented a new direction towards a certification aware Mixed Criticality System (MCS).

2.2.2 Cost Reduction

When various tasks share a CPU for execution, a problem arises, namely, how much execution time should be allocated to each task. Problems pertaining to execution time are traditionally solved by executing tasks on different ECUs, as mentioned in Section 2.2.1. This approach ensures that there is no or controllable interference due to dependencies on tasks that are running on other ECUs. Another nice aspect of this approach is that different companies can develop their own systems independently, with integration only at the level of networks, e.g. CAN. Although proven to be efficient, the approach is not cost effective. Adding more ECUs including cost of cables and connectors adds to the cost of the vehicle multi-fold. Furthermore, it increases the Number of Points of Failure, which is not desirable as mentioned before.

Another solution is to allow different tasks to share one ECU. Though multiplexing tasks on one ECU reduces cost, sharing of resources causes temporal and spatial problems. As described earlier, these tasks may have different criticality and are viewed differently by the CAs. The tasks are required to be verified for different levels of assurances called *Criticality Levels*. When tasks conforming to various Criticality Levels execute together, another class of issues arise that pertain to Mixed Criticality, namely, issues related to guaranteeing the deadlines of high-criticality tasks. When CPU time sharing is taken into account, it therefore becomes important to provide additional run-time support required to meet the deadline constraints of (at least) the highly critical tasks. The issues that arise due to certification and cost-reduction can be precisely illustrated with help of the example given in [13]

Example to illustrate the impact of Certification and Cost reduction on analysis and schedulability. Quote from [13]

Consider a system to be implemented on a preemptive fixed priority uniprocessor, that comprises just three jobs: J1, J2, and J3. All three jobs are released at time zero. Job J1 has a deadline at time-instant 2, while the other two jobs have their deadlines at time-instant 3.5. Jobs J2 and J3 are high-criticality jobs and subject to certification, whereas J1 is a low-criticality job and hence is not.

The system designer is confident that each job has a worst-case execution time (WCET) not exceeding 1. Hence, all three jobs will complete by their deadlines as long as J1 is not given the lowest priority.

However, the CA requires the use of more pessimistic WCET estimates during the certification process, and allows for the possibility that jobs J2 and J3 may each need 1.5 time units of execution. Even if J1 executes for only 1 time unit, jobs J2 and J3 are only schedulable if they are given the highest two priority levels; but if this is done J1 will miss its deadline even if J2 and J3 only execute for 1 time unit.

One might therefore conclude that the system cannot be scheduled in a manner acceptable to both the system designer and the certification authority. Fortunately there is an implementation scheme (and priority assignment) that can satisfy both parties. Consider the priority ordering J2, then J1 and finally J3; and the following run-time behaviour from time 0:

- Execute the highest priority job J2 over [0, 1).
- If J2 completes execution by time-instant 1, then execute J1 over [1, 2) and J3 over [2, 3), thereby ensuring that all deadlines are met (J3 could therefore execute over [2, 3.5) if needed).
- If J2 does not complete execution by time-instant 1, then discard J1 and continue the execution of J2, following that with the execution of J3 over [1.5, 3).

So if the system designer is right, all jobs execute for 1 time unit and meet their deadlines. If the CA is right, then the two high-criticality jobs execute for 1.5 time units each and meet their deadlines. Both parties are satisfied

In this report, we will see if we can improve the performance with the help of software and address the design-related issues of an MCS with the help of MC models discussed in Chapter 3.

2.2.3 Runtime Robustness

The third key driver for MC scheduling is Runtime Robustness. In computer science, robustness is the ability of a computer system to cope with errors during execution [5]. Baruah *et al.* [13] highlight that run-time

robustness is a form of fault tolerance. It allows graceful degradation to occur in a manner that is mindful of criticality levels. Informally speaking, in the event that not all tasks can be serviced satisfactorily, the goal is to ensure that higher criticality tasks shall still be guaranteed, potentially at the cost of no longer guaranteeing lower criticality tasks.

The related terms are:

Fault-Tolerance

Fault tolerance, or graceful degradation, is the property that enables a system to continue operating properly in the event of the failure of (or one or more faults within) some of its components. If its operating quality decreases at all, the decrease is proportional to the severity of the failure, as compared to a naïvely designed system in which even a small failure can cause total breakdown. Fault tolerance is particularly sought after in high-availability or life-critical systems [2].

Resilience

Within the real-time systems domain, the notion of (fault) resilience has been described as the ability to tolerate (i.e. recover from) errors while remaining schedulable [19]. Stated differently, the real-time deadlines of the system can still be met.

In case of several tasks running on one CPU, it becomes important to address the issues related to inter-application and intra-application robustness. This means that faults of one task should not impact other tasks within the application (intra-application). Similarly, faults of one application should not impact the execution of the other applications (inter-application). A technological advancement, in the domain that helps resolve these issues of inter-application robustness, is Hardware Virtualization. To resolve issues related to intra-application robustness, run-time monitoring is used.

Run-time Monitoring

In the Real-Time domain, Temporal Isolation has always been a key driver for run-time monitoring, when multiplexing applications on a shared CPU is envisioned. Hierarchical scheduling and CPU reservation are examples of the same. In both approaches, monitoring plays a vital role in detecting whether the allocated execution time budget has finished. When the budget is finished, actions can be taken to prevent tasks or applications from claiming resources further.

Recently, within the MC context, monitoring has proved to be a quintessential element for detecting tasks exceeding their execution time allowance. Once the monitor has played its role, i.e. detected the task exceeding its execution time allowance, it becomes necessary to prevent fault cascading (Temporal Isolation) and to provide a higher degree of assurance against such faults (Mixed Criticality). It will be difficult, if not impossible, to separate the two. Examples where providing both temporal isolation and criticality level-up changes are necessary is discussed in Chapter 5.

In order to prevent damage, measures/policies have to be adopted for temporal isolation, and criticality level changes appropriately. Some of the policies described in literature are:

Task reset is a viable option if a group of tasks perform a function together. Usually, such tasks are periodic in nature and release smaller independent execution chunks called *Jobs*. Aborting a job in execution and requesting this task to reset to the initial state (provided task recovery routines are available) may ensure correct execution of the next periodic job.

Task abortion is an option commonly used when tasks are independent and do not produce results collectively but individually perform the assigned functionality. In such a case, the erroneous task can be completely aborted and the next periodic releases be de-scheduled as well. In other words, in this case it is not scheduled for execution any more.

Task suspension is used when the state of a task is important and re-initializing the task state is not preferred. Suspending the task ensures that the current state is stored and at a later state the task is resumed. The task can be resumed when sufficient execution time is available.

Task re-prioritization can be used when some progress is expected even in case of a task exceeds its WCET (Overrun). This means, if tasks are working collectively and other tasks depend on the task that exceeds its WCET, the erroneous task may still be allowed to run, at lower priority, in order to enable dependant tasks to make progress. Task re-prioritization can be achieved in many ways like executing the task in the background or reserving a low priority band for erroneous tasks.

These policies can be used in an MCS for temporal isolation of higher criticality tasks against faults of lower criticality tasks, during a criticality level change and otherwise. In Chapter 5, changing the level of assurance (criticality level) is discussed to make the contextual use of these policies explicit. In this report, we intend to use the same names for the policies above although the motivation and semantics may differ from what is described above.

Hardware Virtualization

As mentioned before, Hardware Virtualization helps achieve inter-application robustness.

Computer hardware virtualization is the virtualization of computers or operating systems. It hides the physical characteristics of a computing platform from users, by showing another abstract computing platform [3].

Hardware virtualization can be achieved with the help of a Hypervisor or a Hierarchical Scheduling Framework (HSF). Although there is no strict boundaries to the functionality provided by a hypervisor and an HSF the terms are used contextually in literature. A hypervisor manages the execution of guest operating systems allowing various operating systems to share the same hardware. It is installed either on a host operating system or as a firmware. An HSF is typically used to manage execution at application level allowing various applications to share resources. It is generally installed on one operating system to provide scheduling possibilities to applications with varying temporal requirements.

Hierarchical Scheduling (HS) is a way of realizing virtualization and was first introduced by Deng and Liu [20]. There are various definitions of Hierarchical Scheduling (HS) but the core concept remains the same. Two of the definitions are:

The hierarchical scheduling framework has been introduced to enable compositional schedulability analysis of systems with real-time constraints to simplify schedulability analysis of complex systems [32]. Hierarchical scheduling in RT systems introduces time partitioning of the resources (typically processor) and tries to address the integration and predictability related issues [7].

In HS, as the name suggests, there is multilevel scheduling. In a typical HSF there are schedulers at two levels. A *Component* in HSF is defined by a set of tasks which are scheduled by a local scheduler. This component can be referred to as an application. A *Server* defines the component's time budget (i.e. its share of the processing time) and its replenishment policy. Few advantages of HS, which also present the core rationale behind its introduction, are (among others):

- Temporal isolation/partitioning among applications (adds to runtime robustness of the system)
- Individual subsystem-based analysis, design and testing
- Ease of integration from a timing perspective

Using an HSF on a powerful CPU will lead to cost reduction as well since the applications would otherwise execute on different ECUs. As various applications run in an automotive system, as described above, it will be highly beneficial to use an HSF given the aforementioned advantages. Using an HSF, various applications can be compartmentalized into the HSF's components leading to inter-application robustness.

To conclude, as various applications run in an automotive system it will be highly beneficial to compartmentalize these applications using an HSF, given its advantages. Using an HSF, various applications can be allowed to share a platform thereby reducing cost. In our context, platform refers to a CPU. Temporal fault detection and actions for run-time support (for efficiency reasons) are necessary to provide certifiable assurances to safety and mission critical tasks over non-critical tasks.

2.3 Derived application drivers

The application drivers (refer Figure 2.4) derived from the non functional requirements of the automotive domain are MC and HSF.

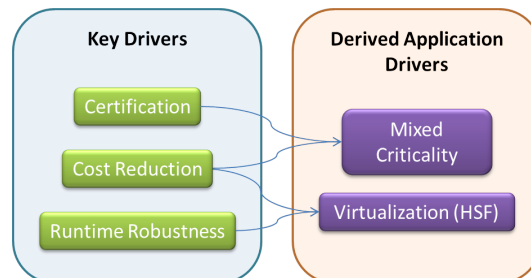


Figure 2.4: Key Drivers revisited

In this report, various MC schemes, discussed in literature, are summarized. Then, we present a generalization of a simple MC scheme. We present the design and implementation of the scheme and extend a real-time scheduler to support Mixed Criticality.

Chapter 3

Preliminary Models and Definitions

Towards a more detailed problem analysis, in this chapter, we summarize some of the mixed criticality scheduling schemes described in the literature. The objective is to analyse these schemes, so that it leads us towards the specifications to realize an MCS. In particular, we show how different schemes define and detect temporal faults. Once a fault is detected, the fault is handled by the policies defined in the individual schemes. Ensuring safe execution of at least high(er)-criticality tasks comes potentially at the cost of low(er)-criticality tasks missing their deadlines. We are therefore interested in the conditions for criticality level changes and the proposed mechanisms for handling them.

In the next section (Section 3.1), a generic MC task model is presented. Using the generic model, various approaches, described in literature, are discussed in the sections 3.2, 3.3, 3.4 and 3.5. An overview is presented in Section 3.6.

3.1 Generic Mixed Criticality Model

The system must support a set of criticality levels represented as:

1. $\mathcal{L} = \{L^1, L^2, \dots, L^N\}$ with $N \in \mathbb{N}^+$. \mathcal{L} is an ordered set of criticality levels where L^N is the highest criticality level.

A generic MC task model can be expressed as:

2. A set of n independent periodic tasks, τ_i for $i = 1..n$
 3. $\rho_i \in 1..n$: unique priority of τ_i where n is the highest priority
 4. T_i : period of τ_i
 5. D_i : deadline of τ_i
 6. $\forall l \in \mathcal{L}$: $C_i(l)$ is the WCET for τ_i at criticality level l .
 7. L_i : representative criticality level for τ_i . The representative criticality level is the highest level of assurance required by a task to complete its execution.
 8. β_i : the actual execution time of the current job of task τ_i
 9. l : the system's current criticality level
 10. l^{new} : the new higher criticality level in which the system will execute after a criticality level change
- The following constraints and assumptions apply:
11. $D_i \leq T_i$, i.e. a constrained deadline system

$$12. \forall i, l: (l \leq L_i \rightarrow C_i(l) \leq C_i(L_i))$$

The approach to allocate higher priorities to higher criticality tasks may not lead to an optimal CPU utilization. Since tasks of various criticality levels will compete together, the system must be able to cope with the temporal constraints set by the CA. The level of assurances (execution time) required by the tasks dictate the number of criticality levels that the system should support. Therefore, to optimize the CUP utilization and to provide higher assurances to tasks, the criticality level of the system can be changed to a higher criticality level at runtime. As mechanisms of error/fault recovery are desirable, lower criticality tasks must be allowed to execute by changing the criticality level of the system to a lower criticality level at a safe moment.

The following condition defines when a level-up change is made.

Condition 1 (Criticality level-up change condition). *When a job of task τ_i is executing at time t while the system is running at criticality level $l | l \leq L_i$ and $\beta_i(t) = C_i(l)$, a criticality level-up change occurs.*

The Condition 1 means that when the system is running at a criticality level L^1 ($l = L^1$) and a job of a task τ_i exceed its allowed execution time of $C_i(L^1)$, a criticality level-up change occurs. When the current criticality level l of the system is L^2 , the condition dictates that no tasks with representative criticality level L^2 are allowed to execute until it is safe to do so.

We now define subsets \mathcal{T}^l of \mathcal{T}

$$\mathcal{T}^l \stackrel{\text{def}}{=} \{\tau_i | L_i \geq l\} \quad (3.1)$$

A subset of lower criticality tasks can be defined as:

$$\mathcal{T}^{LOW}(l) = \{\tau_j | L_j < l\} \quad (3.2)$$

A level- l idle time is defined as follows.

Definition 1 (Criticality level- l idle time). *A criticality level- l idle time is an instant at which there is no pending load of the tasks in \mathcal{T}^l , i.e. there are no pending jobs by tasks in the set \mathcal{T}^l .*

The following condition defines when a level-down change is made.

Condition 2 (Criticality level-down change condition). *Upon a criticality level- l idle time a level-down change is performed.*

All MC schemes discussed in this chapter adhere to the model presented above apart from variations in 1 and 12. These schemes also differ in the assumptions they make, in particular, individual models differ in the following assumption.

Assumption 1 (Assumption). *Execution times for τ_i will not exceed $C_i(L_i)$*

The condition for a criticality level-up change is common to all the relevant models. The notations used in this report may differ from the notations used in the respective papers. Finally, the papers also differ in the *policies* after a level-up or level-down change is made, e.g., whether lower-criticality tasks are allowed to run.

3.1.1 Vestal's Scheme - 2007

As described in Vestal's paper [33], a deadline monotonic priority assignment leads to a problem called *Criticality Inversion*, i.e., a lower-criticality higher-priority task can block a higher-criticality lower-priority task. His main contribution was a priority ordering algorithm for scheduling a Mixed Criticality task set. A technique called Period Transformation was used to overcome the criticality inversion. Vestal also proposed using Audsley's algorithm as an alternative to or in combination with period transformation.

The variations of Vestal's model [33] from the generic model presented in Section 3.1 are:

1. $\mathcal{L} = \{L^1, L^2, L^3, L^4\}$ an ordered set of four criticality levels, where L^4 is the highest criticality level.
12. $\forall i: C_i(L^1) \leq C_i(L^2) \leq C_i(L^3) \leq C_i(L^4)$

Vestal does not follow the Assumption 1. A criticality level-up change follows Condition 1 and a criticality level-down change was not discussed in this paper.

It is important to note here that lower-criticality tasks are also analysed for their computation times at higher criticality levels. Hence, higher values of WCETs are also allocated to low-criticality tasks at criticality levels higher than their representative criticality levels.

3.1.2 Baruah and Vestal - 2008

Baruah and Vestal [12] add to Vestal's original scheme by explicitly allowing unknown WCET estimates for lower criticality tasks at criticality levels higher than L_i . Their argument for this is that low-criticality tasks shouldn't need to adhere to the strict standards that high-criticality tasks need to adhere to. They also mention that, by setting these WCET values at higher criticality levels to ∞ , these lower criticality tasks may be allowed to run but their respective deadlines cannot be guaranteed. The variation of their model can be expressed as:

1. $\mathcal{L} = \{L^1, L^2, \dots, L^N\}$ with $N \in \mathbb{N}^+$. \mathcal{L} is an ordered set of criticality levels where L^N is the highest criticality level.
12. $\forall i, l: ((l \leq L_i \rightarrow C_i(l) \leq C_i(L_i)) \wedge (l > L_i \rightarrow C_i(l) = \infty))$

Therefore, in [12], Assumption 1 is not made. Condition 1 is used as the criterion for a criticality level-up change and a criticality level-down change is not discussed.

To achieve temporal isolation, they assign priorities to low-criticality tasks in levels higher than L_i so that these low-criticality tasks do not interfere with safe execution of higher criticality tasks. One way of achieving this is to execute these low criticality tasks in the background. It is important to note here that, initially (upon system initialization) the priorities are statically assigned according to Vestal's algorithm. A level-up change may lead to priority reordering in order to ensure safe execution of high-criticality tasks. In the paper, only policies for a level-up change (called overrun in their terminology) are defined. It follows that monitoring of all tasks is essential to prevent a task from exceeding its allowed $C_i(l)$ value. The event of a task exceeding its allowed $C_i(l)$ value is described as the key factor for a criticality level-up change.

3.2 SMC Scheme - 2011

The Static Mixed Criticality scheme of Baruah *et al.* [11] and Vestal's original scheme both require run-time monitoring of individual tasks to detect a task exceeding its allowed $C_i(l)$ value triggering a criticality level-up change. The difference between the two schemes is that in Vestal's original model, lower criticality tasks are also analysed for their computation times at higher criticality levels. As mentioned earlier, to verify low-criticality tasks for the same level of assurance as high-criticality tasks is an expensive approach in terms of timing analysis and resource usage; SMC proposes the solution as an extension.

The variation on the generic MC task model can be expressed as:

1. $\mathcal{L} = \{L^1, L^2\}$ is an ordered set of two ($N = 2$) criticality levels, where L^2 is the highest criticality level

$$12. \forall i, l: ((l \leq L_i \rightarrow C_i(l) \leq C_i(L_i)) \wedge (l > L_i \rightarrow C_i(l) = C_i(L_i)))$$

In the SMC scheme, two modes are defined, a Normal Mode and an Overrun Mode. We will describe the behaviour of this scheme with respect to the model presented in Section 3.1. At the systems's current criticality level L^1 all tasks, low-criticality as well as high-criticality tasks, run at priorities assigned by Vestal's algorithm. When any task of representative criticality level L^1 executes more than its allowed WCET while the system is running at criticality level L^1 , a level-up change is executed. The system then starts executing at criticality level L^2 . At criticality level L^2 , all the low-criticality tasks are assigned a priority lower than any high-criticality task, making sure that no high-criticality task is impacted.

In the paper, Condition 1 is not defined, rather Assumption 1 is lifted and called *Overrun*. The paper describes (for the first time) an idle time as an appropriate moment for going back to criticality level L^1 and allowing low-criticality tasks to execute at their normal priorities again. If $l = L^2$, detection of a level- l idle time is the trigger for going back to a lower criticality level.

3.3 AMC Scheme - 2011

Yet another but more strict extension of Vestal's scheme is the AMC scheme by Baruah *et al.* [13]. For simplicity, the scheme was described for a dual criticality system but is also valid for a multi-criticality system. It is proven to be better than SMC in terms of schedulability which comes at a cost of all lower criticality tasks being aborted rather than allowing them to execute at lower priorities. Baruah *et al.* also showed in this paper that Vestal's priority assignment algorithm is applicable to the AMC scheme as well.

In the AMC scheme, once any task does not finish within its allowed execution time, the criticality level is increased and none of the lower criticality tasks are allowed to execute. The extension can be represented in terms of the generic MC task model as:

1. $\mathcal{L} = \{L^1, L^2\}$ is an ordered set of two ($N = 2$) criticality levels, where L^2 is the highest criticality level

$$12. \forall i, l: ((l \leq L_i \rightarrow C_i(l) \leq C_i(L_i)) \wedge (l > L_i \rightarrow C_i(l) = 0))$$

The AMC scheme does not make Assumption 1, and Condition 1 is used as the criterion for a criticality level-up change. Likewise, Condition 2 is used as a criterion for a criticality level-down change. Note that the policy for handling the criticality-level change is different in the AMC scheme. In the AMC scheme, tasks are not allowed to run upon a criticality level-up change whereas in the SMC scheme they continue to execute but at lower (background) priorities such that they do not interfere with high-criticality tasks.

3.4 Relaxed-AMC Scheme - 2012

The schedulability of the AMC scheme was an improvement over SMC, since the former does not allow any lower criticality task to execute at higher criticality levels. Other improvements were proposed by Santy *et al.* [31] to allow some lower-criticality higher-priority tasks to run at higher criticality levels. Santy *et al.* relaxed some of the assumptions of the AMC scheme. To express their assumptions in the generic MC task model:

1. $\mathcal{L} = \{L^1, L^2, \dots, L^N\}$ with $N \in \mathbb{N}^+$. \mathcal{L} is an ordered set of criticality levels where L^N is the highest criticality level.

$$12. \forall i, l: ((l \leq L_i \rightarrow C_i(l) \leq C_i(L_i)) \wedge (l > L_i \rightarrow C_i(l) = C_i(L_i)))$$

Santy *et al.* strictly follow Assumption 1, and in case the assumption fails they call it erroneous behaviour. A criticality level-up change does NOT follow Condition 1. In this scheme a criticality level-up change is performed when a task τ_i exceeds its WCET $C_i(l)$ while the system is running at criticality level $l | l < L_i$. However, the level-up change can be delayed under certain conditions. For this, they introduced the concept of allowance. The design and implementation of allowance is kept out of scope of this report.

They proved that even at higher criticality level low-criticality high-priority tasks can be allowed to execute, unless they exceed their $C_i(L_i)$ WCET. Another contribution is that they proved that an idle time is an appropriate moment to go back to a lower-criticality level. The criticality level-down change is performed when Condition 2 occurs.

3.5 Practical MC Scheme - 2013

In [15], Burns *et al.* propose a practical implementation of MCS in the context of a dual criticality system. The adaptations can be expressed as:

1. $\mathcal{L} = \{L^1, L^2\}$ is an ordered set of two ($N = 2$) criticality levels, where L^2 is the highest criticality level
12. $\forall i: C_i(L^1) \leq C_i(L^2)$

They also lift Assumption 1. In this scheme if a task exceeds its WCET at its representative criticality level it is suspended until its next release. The semantics of suspension are not clear from the paper. A criticality level-up change is made under the same conditions as in the Relaxed-AMC scheme. They add that at criticality level $l = L_i$, a task τ_i executing longer than its $C_i(L_i)$ WCET is suspended. A criticality level-down change is made under the same conditions as in the SMC scheme, i.e., upon an criticality level- l idle time.

The main characteristics of their proposal are:

1. Re-prioritize the low-criticality tasks in high-criticality mode such that it ensures safe execution of high-criticality tasks. Note that the low-criticality tasks may not meet their deadlines in the high-criticality level.

$$\forall l \in \mathcal{L} | l > L_i : C_i(l) = \infty$$

2. Reduce the WCET of low-criticality tasks in the high criticality level. This means a low quality of service.

$$\forall l \in \mathcal{L} | l > L_i : C_i(l) < C_i(L_i)$$

3. Increase the period of low-criticality tasks in high-criticality levels so that they execute less often, hence a low quality of service.

$$\forall l \in \mathcal{L} | l \leq L_i : T_i(l) = T_i(L_i)$$

$$\forall l \in \mathcal{L} | l > L_i : T_i(l) > T_i(L_i)$$

4. Exploit so-called gain time to allocate more CPU time to low-criticality tasks, where gain time is the surplus time when high-criticality jobs finish in less than their level l WCET.

They also propose a robust priority assignment scheme which can be applied to Santy *et al.*'s work.

3.6 Overview

A comprehensive overview of the following schemes is presented in Table 3.1, using the generic MC task model.

- Vestal's Scheme - 2007[33]
- Baruah and Vestal - 2008[12]
- SMC Scheme - 2011[11]
- AMC Scheme - 2011[13]
- Relaxed-AMC Scheme - 2012[31]
- Practical MC Scheme - 2013[15]

To summarize, in an MCS a level-up change generally occurs upon Condition 1 and level-down change, depending on the scheme, happens upon idle time. Further in this report we consider an extension of the AMC scheme.

Table 3.1: Comparison of MC schemes

Scheme	Assumptions	Priority assignment	WCET allocation where $l > L_i$	Level-up change condition	Level-up change handling					Overrun condition $l \geq L_i$	Overrun handling	Level-down change condition	Comments
					Policy	Higher-Priority Lower-Criticality tasks	Higher-Priority Higher-Criticality tasks	Lower-Priority Lower-Criticality tasks	Lower-Priority Higher-Criticality tasks				
Vestal's Scheme	Task will never exceed $C_i(L_i)$	Audsley's	$C_i(l) \geq C_i(L_i)$	$l \leq L_i$ and $\beta_i = C_i(l)$ and task complete event not signalled	No priority re-ordering	Task continue to run on same priorities	Task continue to run on same priorities	Task continue to run on same priorities	Task continue to run on same priorities	Not defined	Not defined	Not defined	
Vestal and Baruah	Task will never exceed $C_i(L_i)$	Hybrid-priority scheduling (Augmented Audsley's)	$C_i(l) = \infty$	$l \leq L_i$ and $\beta_i = C_i(l)$ and task complete event not signalled	re-prioritize such that τ_i does not interfere with high criticality tasks	- Run at re-ordered(to lower) priorities. - No deadline guarantees	-Run at same/re-ordered(to higher) priorities. - With deadline guarantees	- Run at same/re-ordered(to lower) priorities. - No deadline guarantees	- Run at re-ordered(to higher) priorities. - With deadline guarantees	Not defined	Not defined	Not defined	
SMC		Initially Audsley's and criticality aware priority assignment upon level-up change	$C_i(l) > C_i(L_i)$ also called overrun budget	$\beta_i = C_i(l)$ and task complete event not signalled	re-prioritize such that τ_i does not interfere with high criticality tasks	- Run at re-ordered(to lower) priorities. - No deadline guarantees	-Run at same/re-ordered(to higher) priorities. - With deadline guarantees	- Run at same/re-ordered(to lower) priorities. - No deadline guarantees	- Run at re-ordered(to higher) priorities. - With deadline guarantees	Not defined	Not defined	Criticality level- l idle period	
AMC	Task will never exceed $C_i(L_i)$	Audsley's	$C_i(l) = 0$	$l \leq L_i$ and $\beta_i = C_i(l)$ and task complete event not signalled	- No priority re-ordering - All tasks in $\mathcal{T}^{LOW}(l)$ are not allowed to run.	Do not run	- Run at same priorities - With deadline guarantees	Do not run	- Run at same priorities - With deadline guarantees	Not defined	Not defined	Criticality level- l idle period	
Relaxed-AMC		Audsley's	$C_i(l) = C_i(L_i)$	$l < L_i$ and $\beta_i = C_i(l)$ and task complete event not signalled	No priority re-ordering	- Run at same priorities - With deadline guarantees	- Run at same priorities - With deadline guarantees	- Run at same priorities - No deadline guarantees	- Run at same priorities - With deadline guarantees	Not defined	Not defined	Criticality level- l idle period	Criticality level-up change delayed using ℓ -allowance
Practical MC Scheme		Initially Audsley's, then more criticality aware priority reordering (statically assigned) upon a level-up change	1) When priority re-ordering is allowed: a. $C_i(l) = C_i(L_i)$ or b. $C_i(l) = \infty$ 2) When priority reordering is NOT desirable: a. $C_i(l) < C_i(L_i)$ by using gain time or b. $C_i(l) = 0$	$l < L_i$ and $\beta_i = C_i(l)$ and task complete event not signalled	- Subject to WCET allocation at $l > L_i$ - Conforms with schemes mentioned above	- Subject to WCET allocation at $l > L_i$ - Conforms with schemes mentioned above	- Subject to WCET allocation at $l > L_i$ - Conforms with schemes mentioned above	- Subject to WCET allocation at $l > L_i$ - Conforms with schemes mentioned above	- Subject to WCET allocation at $l > L_i$ - Conforms with schemes mentioned above	if $\beta_i = C_i(L_i) \wedge$ task complete event not raised	Suspended until next release	Criticality level- l idle period	At $l > L_i$ - Proposed changing task period - Capacity inheritance with budget reduction scheme suggested

Chapter 4

System Requirements

This chapter describes the requirements for the AMC scheme. There was no official requirements document for the assignment, other than the goals for i-Game project. This is due to the research nature of the assignment. Hence, a significant amount of time was invested in reading the relevant literature from which the requirements of the system are derived. The influencers for deriving the requirements for the assignment are the key drivers of Mixed Criticality scheduling, as discussed in Chapter 2. The main sources of requirements for MC scheduling are the MC schemes presented in the Chapter 3.

One of the task (amongst others), of WIN-SAN group, in WP2 of the i-Game project, is to address the mechanisms required for criticality-level changes on *(i)* a single-core (X86-based) and *(ii)* a multi-core hierarchically scheduled platform. These mechanisms should then be extended to support for resource sharing. Considering the broad spectrum of the problem domain the scope of the PDEng assignment was narrowed down to designing a mechanism for Criticality-Level changes on a single-core platform keeping the aspects of hierarchical scheduling for future work. No sharing of resources other than the CPU will be taken in to account.

For prototyping purposes, for the assignment it has been decided to use:

- Linux as an operating system.
- A single core desktop-grade processor.
- Fixed Priority Task Scheduling.

4.1 Requirements for the AMC scheme

In this section we describe the requirements for the AMC scheme.

- The system must define the number of criticality levels supported.
- It must be possible to associate a criticality level to each task.
- A task must be associated with a WCET per criticality level.
- The developer must associate a task with one criticality level-change policy.
- The system must be able to monitor the actual execution time of a task.
- Based on monitoring, the system must identify when criticality level changes are needed. For this, tasks executing longer than their $C_i(l)$ execution time need to be detected.
- System must be capable of switching criticality levels.
 1. Low criticality level to high criticality level.

2. High criticality level to low criticality level.
 3. Multiple criticality level changes.
- The system must provide a handler for:
 - The level-up trigger.
 - The level-down trigger.
 - For each handler there must be a default policy.

4.2 Level-up trigger handler

- Assigning new values of WCET to tasks, according to system's new criticality level.
- Task suspend.
- Job abort.

4.3 Level-down trigger handler

- Assigning new values of WCET to tasks, according to system's new criticality level.
- Task resume.

4.4 Domain specific requirement

A requirement of Mixed Criticality scheduling is that the overhead of the criticality level-up handler should be as low as possible.

Chapter 5

Scheme Semantics

In this chapter we present the semantics of the AMC* scheme which is an extension of the AMC scheme. For this scheme, we show how the criticality-level-change triggers are identified and which actions the system has to execute upon criticality-level changes. Run-time monitoring forms the basis for all the schemes discussed in Chapter 3. In the next chapter, we come back to providing support for run-time monitoring. In this chapter, we discuss the criticality-level changes in Section 5.1. Then, we discuss the expected system behaviour in Section 5.2. In Section 5.3 we discuss the cases of criticality-level changes. Finally, we present six scenarios in Section 5.4 which illustrate the expected system behaviour.

5.1 AMC* Scheme

In this section, we discuss the conditions for criticality-level changes. First, the policies adopted upon criticality-level changes are presented. Then, we discuss the criticality level-up change followed by the criticality level-down change. We assume constrained deadlines of tasks; see item 11 in Section 3.1.

We adopt the following definition from the paper in which the AMC scheme [13] was proposed. In this definition, C is a vector of WCETs indexed by criticality level.

Definition 1 (Behaviors). *"During different runs, any given task system will, in general, exhibit different behaviors: different jobs may be released at different time instants, and may have different actual execution times. Let us define the criticality level of a behavior to be the smallest criticality level such that no job executed for more than its C value at this criticality level."*

The AMC scheme [13] was described for a system supporting tasks of two criticality levels. We call the AMC scheme described here the AMC* scheme, because the AMC scheme does not **explicitly** describe situations where multiple criticality level changes may have to be executed, in case the system supports more than two criticality levels. The semantics of the AMC* scheme specializes to the AMC scheme in [13] for $N = 2$ criticality levels. From [13], the semantics for handling the task which causes a criticality level-up change must be refined. Particularly, the policies deployed to stop the tasks from contending for resources are unclear. These are not important for the timing analysis but they are important from an implementation perspective.

According to the AMC scheme, the system always initializes the current criticality level l to the lowest criticality level L^1 . The initial priority of each task is assigned according to Vestal's algorithm [33]. From the semantics described in [13], a criticality change is triggered when a task τ_i did not complete within its $C_i(l)$ WCET, i.e., the actual execution time β_i of the current job of τ_i is equal to $C_i(l)$. First, we define the policies which are adopted upon a criticality-level change.

Definition 2 (Suspend). *The Suspend policy for a task (i) does not give any more execution time to a currently active job, and (ii) suppresses new releases of jobs.*

Definition 3 (Abort). *The Abort policy decides if the current job of a suspended task is discarded or allowed to continue from where it stopped.*

The policy to actually abort a job of a suspended task is appropriate for the core automotive tasks since a delayed response has no value. Therefore, jobs of suspended tasks are always aborted for AMC*. For example, if a periodic task that reads the value of the GPS sensor is suspended, upon resumption it becomes important to get the updated location from the GPS sensor. The last sensor reading is not relevant any more. Hence, the current job of the suspended task can be aborted. The realization of the Abort policy is described in the implementation chapter, i.e., Chapter 8.

Definition 4 (Resume). *The Resume policy at time t_r means that a suspended periodic task τ_m is allowed to release a new job at time $a_{m,k}$ (Equation (5.1)).*

If ϕ_m is the activation time of the first job of τ_m , $a_{m,k}$ is given by

$$a_{m,k} = \phi_m + k \cdot T_m \quad (5.1)$$

where k is determined by

$$k = \min\{k | \phi_m + k \cdot T_m \geq t_r\}. \quad (5.2)$$

Now, we look at the triggers for criticality-level changes and the actions to be taken in order to realize these changes.

Level-Up trigger : A criticality level-up trigger happens upon Condition 1. At the system's current criticality level l , if a task τ_i did not complete within its $C_i(l)$ WCET ($\beta_i = C_i(l)$), then the following cases can be distinguished:

Case 1 $l \leq L_i < L^N \wedge C_i(l) = C_i(L_i)$. This case represents a task that does not complete its execution within the WCET at its representative criticality level which is lower than L^N . The new criticality level l^{new} is identified by

$$l^{new} = L_i + 1. \quad (5.3)$$

Case 2 $C_i(l) < C_i(L_i)$. This case represents a task that does not complete its execution within its $C_i(l)$ WCET where l is a criticality level lower than its representative criticality level. The new criticality level l^{new} is identified by

$$l^{new} = \min\{L | C_i(l) < C_i(L)\}. \quad (5.4)$$

Error Condition $l \leq L_i = L^N \wedge C_i(l) = C_i(L_i)$. The error condition represents a case where l^{new} cannot be identified. It is an important condition since one of the highest criticality tasks is not finished, so one may want to handle this situation by logging a trace. Alternatively, the current job of the task may be aborted. The policy to abort the current job allows other tasks to execute. Since aborting a job of the most critical task is undesired and L^N WCETs are the WCETs that the CAs impose, in our semantics we consider that the error condition can not occur.

There are two common attributes in Case 1 and Case 2 mentioned above. The first common attribute is that all tasks in $\mathcal{T}^{l^{new}}$ are allowed to execute for their $C_i(l^{new})$ WCETs. The second common attribute is the way jobs of tasks with a lower criticality level, i.e., $\tau_j \in \mathcal{T}^{LOW}(l^{new})$, are handled. Any active job of τ_j is aborted and the subsequent activations of τ_j are suppressed, i.e., the task is suspended until the system's criticality level is reduced (to be discussed next).

Handling the level-up trigger : Now, corresponding to each case (Case 1 and Case 2), we explicitly highlight the differences in handling the current job of task τ_i , the task that caused the level-up trigger.

Case 1 Task τ_i belongs to $\mathcal{T}^{LOW}(l^{new})$; τ_i is suspended and the current job of the task τ_i is aborted. Care must be taken during the system design phase that the overhead of the job abort policy does not come at the cost of execution time of the tasks in $\mathcal{T}^{l^{new}}$.

Case 2 Task τ_i belongs to \mathcal{T}^{new} and the current job of τ_i is allowed to execute for another $C_i(l^{new}) - C_i(l)$ time units.

The level-up trigger can give rise to an anomaly. Consider a system with two criticality levels L^1 and L^2 and the system's current criticality level is L^1 . Suppose all tasks of criticality level L^2 are complete (i.e., waiting for next job release) and a task of representative criticality level L^1 does not complete within its $C_i(L^1)$ WCET. Then a criticality level-up trigger occurs, therefore all L^1 tasks must be suspended. Note here that since there are no pending jobs of L^2 tasks, the system detects an idle time. Idle time is detected when the number of tasks in the ready queue of the scheduler is zero. Therefore, a criticality level-down trigger follows immediately as a consequence of the level-up trigger. Although refinement of this condition is possible, like allowing the job to continue execution until a job of a higher criticality task is released, we leave this for future work. This anomaly is nicely illustrated by Scenario 6.

Level-Down trigger : The precondition for the level-down trigger is Condition 2. There are two things to decide for a level-down trigger and the level-down change.

1. When the level-down trigger is generated.
2. How many criticality levels to go down.

For these decisions, there are considerations like the cause of the overload condition (e.g. due to the "environment", or the task itself), overheads of the criticality-level changes and ease of implementation.

Options for the level-down change When a level-down trigger occurs, there are various options for the level-down change. Some of the options are:

To the lowest criticality-level : An argument for going to the lowest criticality level is that at a Level- l idle time there is zero **pending** load on the system and the system's state can, therefore, be considered the same as the system's initialization state. An advantage is that this approach is simple from an implementation perspective. The disadvantage is that a possible overload condition due to which the criticality level-up change was performed may not have gone altogether. Hence, there may be a criticality level-up trigger immediately after the level-down trigger has been performed. In this case, the system overheads of going down to the lowest criticality-level and then coming back to a higher criticality-level again may be considerable.

One level at a time : Going down one criticality level at a time will allow the system to validate whether it is safe to go further down in criticality level in the sense that there is no immediate criticality level-up trigger. The overhead of going one level down and immediately coming back to the higher criticality-level is relatively low, compared to going down to the lowest criticality-level. Another advantage is that no execution time is allocated to tasks with representative criticality level lower than $l - 1$ before going further down in criticality level. A drawback of going back one criticality level at a time is that there can be frequent criticality level-up and level-down changes, or even a cyclic loop of criticality level-up and level-down changes.

Last known criticality-level : Going back to the last known criticality level is similar to, and has the same advantages and disadvantages as going down one level at a time. Note here that if a multiple-criticality level-up change is performed (due to Case 2) then this option runs into a higher risk of an immediate criticality level-up change after a level-down change. This is because some tasks, which belong to the skipped criticality-levels, were not allowed to execute. These tasks may also cause an immediate level-up trigger after the level-down trigger is performed.

Validation based : Upon idle time one can compare the actual execution time β of the last job of each task in \mathcal{T}^l with its WCETs at different criticality levels. If all jobs executed for less than their WCETs at some lower-criticality level, the system's criticality level can be reset to this lower level. The advantage of this approach is that there is less chance that an immediate level-up change is needed. The first disadvantage is that, despite the check, it is not certain

that the cause for an overload condition has disappeared. The second disadvantage is that implementation of such a scheme is much more complicated than others.

Whichever option we choose, we cannot precisely determine whether the overload situation does not exist any more; at least when the criticality level-up change was performed due to Case 1.

Handling the level-down trigger : In this report we choose to go back to the lowest criticality level on an idle time. When the system is executing at criticality level $l | l > L^1$, if a level- l idle time is identified the system goes back to criticality level L^1 . This is similar to what is done in the literature (primarily [31]). Hence, upon an idle time, low criticality tasks are again allowed to generate new jobs.

For all suspended periodic tasks we follow the Task Resume policy (definition 4). If the system's current criticality level is l , upon a criticality level- l idle time we need to identify the next activation time for jobs of tasks in $\mathcal{T}^{low}(l)$. If the criticality level- l idle time happens at time t_r , the next activation of the k^{th} job of a suspended task $\tau_m \in \mathcal{T}^{low}(l)$ happens at time $a_{m,k}$ (Equation (5.1)). All the tasks in \mathcal{T}^{L^1} are allowed to execute for their corresponding values of $C_i(L^1)$ WCETs.

5.2 AMC* Scheme: Expected Behaviour

We present the system behaviour with respect to the semantics described for the AMC* scheme. We cover the scenarios with the help of three state diagrams. A task state diagram, a job state diagram and a level-change handler state diagram. We will also cover the cases due to which a level-up and down trigger may arise.

In the state diagrams we assume that:

- Runtime monitoring is a feature of the scheduler. The Monitor explicitly keeps the track of the actual execution time β of each job. The monitor raises an Overrun event in case $\beta_i = \text{WCET}$.
- Since the wait timer is not explicitly modelled, "Wait expired" is a symbolic representation of the end of the "do /wait for next release" action in the Waiting state of the task.
- The "Completed" event in the job state diagram is also a symbolic representation of the end of computation. Notice that the Job Complete event is generated by the job state diagram when the Completed event causes transition J3.

We chose not to explicitly model the scheduler and the monitor either. Since a change in the WCET for a task is essential for monitoring, the scheduler needs to be aware of this change. This is reflected in the Task State Diagram.

5.2.1 Task state diagram

The task state diagram is presented in Figure 5.1. The task state diagram depicts the Execution states as well as change in the WCET of the task. We assume constrained deadline tasks (see item 11 in Section 3.1) so at most one job of a task can be present at any moment in time.

Notice that the event "Allocate WCET" is sent to all tasks, so each task is responsible for setting its WCET to C_{il} .

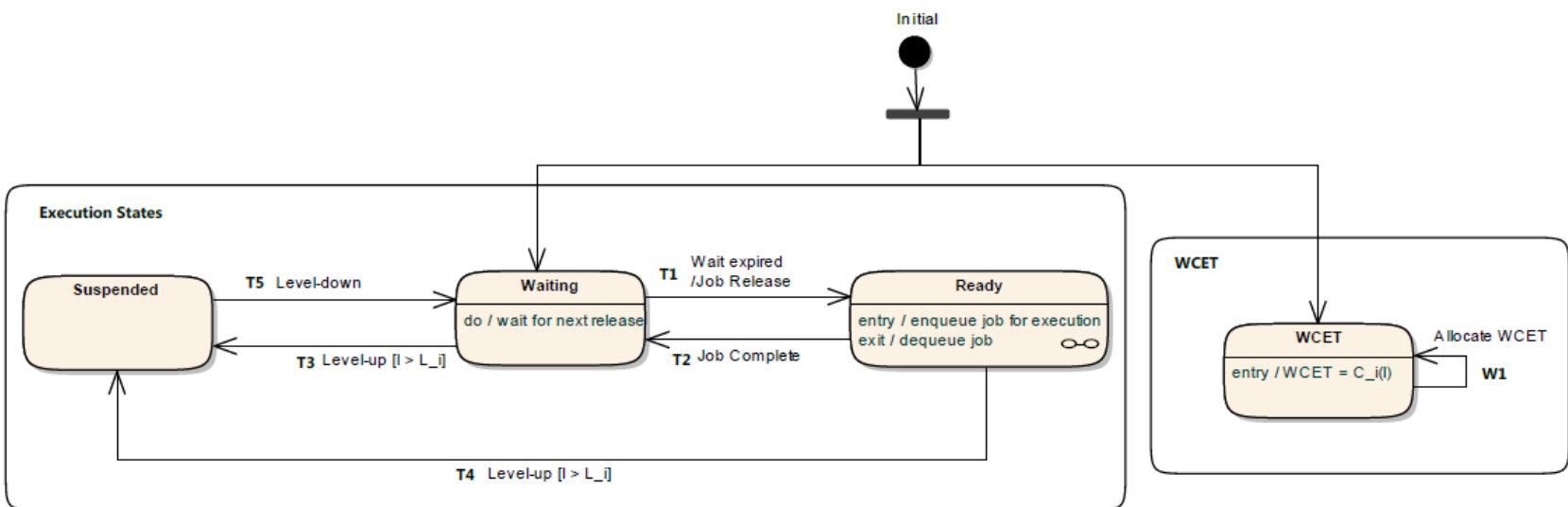


Figure 5.1: Task state diagram.

Task states : A task has the following three states:

Standard task states :

Waiting : A task enters the waiting state upon initialization. The task remains in this state until it releases a new job (according to Equation (5.1)).

Ready : A task enters this state when it releases a new job for execution. As soon as the task enters the ready state its job is enqueued for execution based on its priority. The task remains in this state until its job is complete. Upon completion of the job, the task enters the waiting state again.

WCET : As soon as the task initializes, it enters in this forked state. The task is allocated a WCET corresponding to the system's current criticality level.

Added task state for AMC* :

Suspended : Although the Suspended state is common in modern real-time operating systems, it is added here solely for the purpose of criticality-level changes. A task enters this state only upon a criticality level-up trigger. A task in this state is excluded from using processing resources. None of its jobs is allowed to contend for the CPU. Stated differently: release of jobs of the task are suppressed in this state.

Task state transitions : The following are the state transitions:

Standard transitions :

T1 Waiting to Ready : Wait Expired /Job Release → This transition occurs when the next activation time is reached according to Equation (5.1) and the task releases a new periodic job.

T2 Ready to Waiting : Job Complete → This transition occurs when the released job signals completion.

Added transitions for AMC* :

T3 Waiting to Suspended : Level-up[$l > L_i$] → When a Level-up trigger occurs, all tasks with representative criticality level lower than the new criticality-level of the system are suspended.

T4 Ready to Suspended : Level-up[$l > L_i$] → Similar to T3, all tasks with representative criticality level lower than the new criticality-level of the system are suspended. This transition has an impact on the released job due to the Job Abort policy; this is discussed in next section.

T5 Suspended to Waiting : Level-down → Upon occurrence of a Level-down trigger all tasks in the Suspended state move to the Waiting state.

W1 WCET to WCET : allocate_WCET → This transition occurs when a new WCET must be allocated to the task.

The transitions relevant to the AMC* scene are T3 (Waiting to Suspended), T4 (Ready to Suspended), T5 (Suspended to Waiting) and W1 (WCET to WCET).

5.2.2 Job state diagram

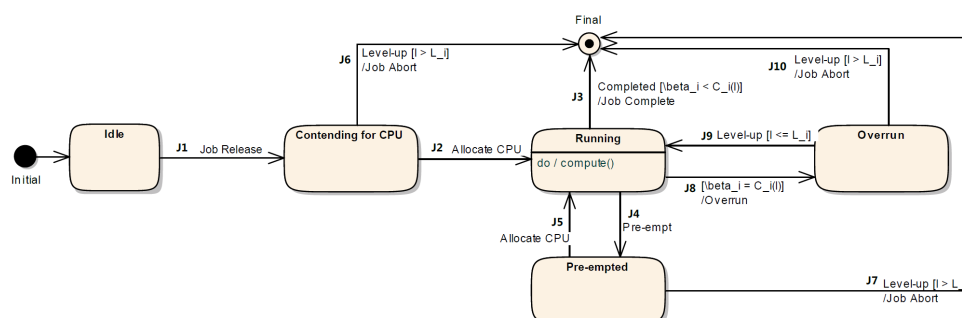


Figure 5.2: A job state diagram of task τ_i .

The job state diagram of task τ_i is presented in Figure 5.2. We distinguish the Contending for CPU state and the Pre-empted state since the actions to be taken for a Pre-empted or Contending job can be different in case of a job abort. In the diagram, β_i (\backslash beta_i) is the actual execution time of the job. The "Completed" event used as a trigger is not a real trigger but a symbolic representation of the job finishing its computation.

The "Overrun" event is actually generated by the Monitor. We have added the Overrun state in the job state diagram because it prevents having to explicitly model the functionality of run-time monitoring.

Job states : A job has the following five states:

Standard job states :

Idle : State before a job is released by a task.

Contending for CPU : A released job first enters this state and waits until the scheduler has executed all higher priority jobs.

Running : A job enters this state when the CPU is allocated. When a job is in the Running state it continues execution until completion. An AMC* scheme-related transition to this state is when a pre-empted job is allowed to continue execution, if the running job is aborted. A second AMC*-scheme associated transition is when a job is allowed to continue execution upon a level-up trigger by a transition from the Overrun state to the Running state.

Pre-empted : A job enters this state if a higher priority job arrives and the scheduler allocates the CPU to that job.

Added job state for AMC* :

Overrun : If a job is not complete within its WCET at the current criticality level, the job enters this state. A new criticality level of the system is identified while the job is in this state. This means that the value of l on exiting this state differs from its value when entering the state.

Job state transitions :**Standard job transitions :**

- J1** : Idle to Contending for CPU : Job Release \rightarrow This transition occurs when a task releases a new job for execution.
- J2** : Contending for CPU to Running : Allocate CPU \rightarrow As soon as all the pending higher priority jobs are executed, the scheduler allocates the CPU to the contending job, which causes this transition.
- J3** : Running to Final : Completed $[\beta_i < C_i(l)]$ /Job Complete \rightarrow When a job is completed and its actual execution time is less than the allowed WCET at the current criticality level of the system, this transition occurs and a Job Complete event is raised. Despite the guards this transition is essentially a standard transition.
- J4** : Running to Pre-empted : Pre-empt \rightarrow When a lower-priority job is in the Running state and a higher-priority job is released, the scheduler allocates the CPU to the higher-priority job. The lower-priority job is pre-empted and this transition occurs.
- J5** : Pre-empted to Running : Allocate CPU \rightarrow This transition occurs when the CPU is allocated to the pre-empted job by the scheduler. This transition gives rise to a special case since it can occur when a lower criticality job in the Running state is aborted, so it is relevant for the AMC* scheme as well. This transition needs to be considered in the scenarios.

Added job state transitions for AMC* :

- J6** : Contending for CPU to Final : Level-up $[l > L_i]$ /Job Abort \rightarrow This transition occurs upon occurrence of a criticality level-up trigger. Only a contending job with representative criticality level lower than the system's current criticality level is aborted.
- J7** : Pre-empted to Final : Level-up $[l > L_i]$ /Job Abort \rightarrow This transition occurs upon occurrence of a criticality level-up trigger. Only a pre-empted job with representative criticality level lower than the system's current criticality level is aborted.
- J8** : Running to Overrun : $[\beta_i = C_i(l)]$ /Overrun \rightarrow If a running job is not complete within its WCET at the system's current criticality level, this transition occurs. The Overrun action is used to trigger a transition to a state in the Level Handler in which the new criticality level is determined. After this, the Level-up event is generated.
- J9** : Overrun to Running : Level-up $[l \leq L_i]$ \rightarrow If the job has a representative criticality level higher than or equal to the system's new current criticality level, this transition occurs and the job is **allowed to continue execution**.
- J10** : Overrun to Final : Level-up $[l > L_i]$ /Job Abort \rightarrow If the job has a representative criticality level lower than the system's new current criticality level, this transition occurs and the job is **aborted**.

Note that when a job is in the Overrun state, a new criticality level of the system is identified hence the value of current criticality level l changes between transition J8 (Running to Overrun) and J9 (Overrun to Running) and between J8 and J10 (Overrun to Final). The transitions relevant to the AMC* scheme are J5 (Pre-empted to Running), J6 (Contending for CPU to Final), J7 (Pre-empted to Final), J8 (Running to Overrun), J9 (Overrun to Running) and J10 (Overrun to Final). When a higher priority job pre-empts the currently running job (J4) and the higher priority job gets aborted (J10), the CPU is allocated again to the pre-empted job (J5). Although the transition J5 (Pre-empted to Running) is triggered by the scheduler, it is relevant for the AMC* scheme when it is a consequence of the transition J10 (Overrun to Final) taken by another job.

5.2.3 Level Handler state diagram

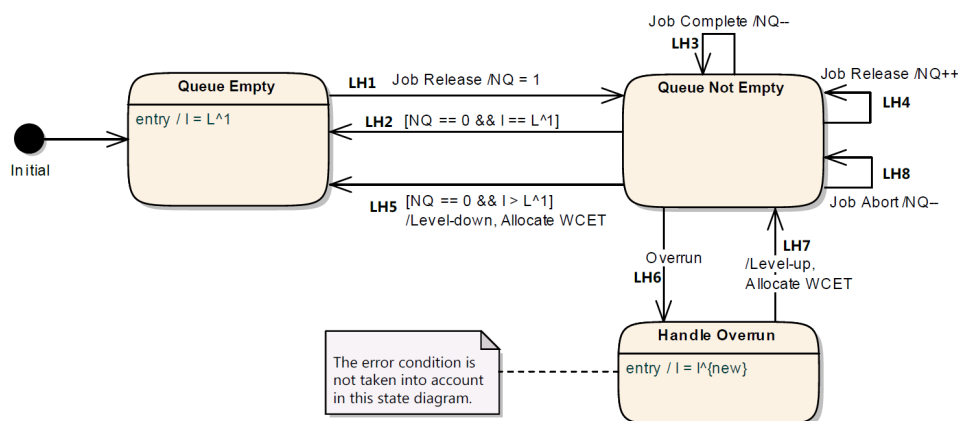


Figure 5.3: A state diagram of the Level Handler. NQ is the number of jobs in the ready queue.

The state diagram of the Level handler is presented in Figure 5.3. When the AMC*-specific states and transitions are deleted, this state diagram models the functionality to signal the occurrence of an idle time; these states and associated transitions are called "Standard" below. An idle time occurs when the ready queue is empty. In Figure 5.3, the number of jobs in the ready queue is denoted by NQ. The Overrun state is added to model how the Overrun event is handled. We assume that simultaneously expiring release timers are handled sequentially.

Level Handler states :

Standard Level Handler states :

Queue Empty : When the ready queue is empty, the current criticality level of the system is always the lowest criticality level, i.e., L^1 . The number of jobs in the queue $NQ = 0$.

Queue Not Empty : When a task releases a job for execution, it triggers the state change to the Queue Not Empty state. In this state the current criticality level of the system could be any criticality level supported by the system. The number of jobs in the queue $NQ \neq 0$.

Level Handler states added for AMC* :

Handle Overrun : The ready queue enters this state when a job is not complete within its $C_i(l)$ WCET. The system's new criticality level l^{new} is identified. The current criticality level of the system l is changed to l^{new} .

Level Handler state transitions :**Standard Level Handler transitions :**

- LH1** : Queue Empty to Queue Not Empty : Job Release /NQ=1 → This transition occurs when a new job is released and the ready queue of the scheduler is empty. The counter for the number of jobs in the ready queue (NQ) is incremented to 1.
- LH2** : Queue Not Empty to Queue Empty : [NQ==0 && $l == L^1$] → This transition occurs when there are no jobs in the ready queue of the scheduler and the system's current criticality level is the lowest (L^1). Despite the $l == L^1$ guard, this is essentially a standard transition.
- LH3** : Queue Not Empty to Queue Not Empty : Job Complete /NQ-- → This transition occurs when a job signals completion; the counter is decremented by one.
- LH4** : Queue Not Empty to Queue Not Empty : Job Release /NQ++ → This transition occurs when a new job is released the counter is incremented by one.

Added Level Handler transitions for AMC* :

- LH5** : Queue Not Empty to Queue Empty : [NQ==0 && $l > L^1$] /Level-down, Allocate WCET → This transition occurs when the ready queue is empty and the current criticality level of the system is higher than L^1 . As a consequence, a Level-down and Allocate WCET triggers are generated.
- LH6** : Queue Not Empty to Handle Overrun : Overrun → When a job is not complete within its $C_i(l)$ WCET, it generates an Overrun trigger (transition J8 in the Job state diagram). As a consequence, this transition occurs and a new criticality level of the system l^{new} is identified.
- LH7** : Handle Overrun to Queue Not Empty : /Level-up, Allocate WCET → This transition occurs once a new criticality level of the system is identified a Level-up and Allocate WCET triggers are generated.
- LH8** : Queue Not Empty to Queue Not Empty : Job Abort /NQ-- → When a job is aborted this transition occurs and the counter for the number of jobs in the ready queue is decremented.

The transitions relevant to the AMC* scheme are LH5 (from Queue Not Empty to Queue Empty), LH6 (Queue Not Empty to Handle Overrun), LH7 (Handle Overrun to Queue Not Empty) and LH8 (a job is aborted). Note that the transition LH5 may occur due to LH8 and due to LH3 (a job completes). The scenarios must consider both situations.

5.2.4 Sequence diagram representing state changes

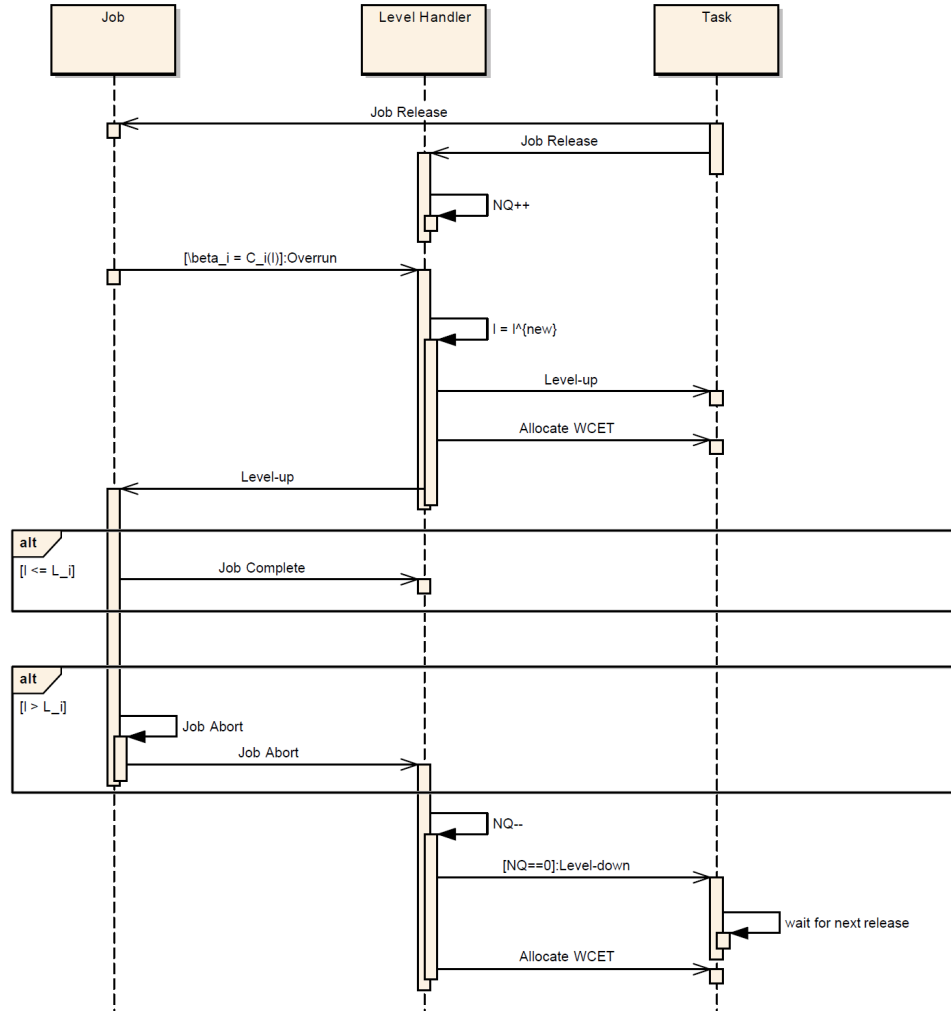


Figure 5.4: Sequence diagram representing the triggers and actions causing the state changes in the three state diagrams.

The sequence diagram in Figure 5.4 depicts the triggers and the actions causing state changes in the state diagrams. Upon a job release by a task, the job makes a transition to the Contending for CPU state and the ready queue counter is incremented.

When an Overrun trigger is raised, the Level Handler identifies the new criticality-level of the system and generates the level-up trigger. The Level Handler triggers the Allocate WCET event as well. The level-up trigger is handled by the task and the job. All tasks with representative criticality level less than the system's new criticality level are suspended. The job of a task τ_i is allowed to continue execution if $l \leq L_i$. In case $l > L_i$, the job is aborted.

When a job either completes execution or is aborted, the number of jobs in the ready queue is inspected. If the ready queue is empty, a level-down trigger is raised. The level-down trigger is handled by a task. The suspended tasks start waiting for the next job release. All tasks are allocated corresponding $C_i(L^1)$ WCETs.

5.3 Cases for the Level-up trigger

We have identified the transitions that need to be included in the scenarios. However there are some special cases that need special consideration namely a "Case 2" (see Section 5.1) multiple-level-up trigger and the "Error Condition" which is not described by the state machines. Now we look at the cases causing a level-up trigger. We look at the cases to derive two scenarios which are not covered yet.

Case 2 : A task not complete within its WCET at a criticality level lower than its representative criticality level (Case 2). This case has two sub cases, a single criticality-level change and a multiple criticality level change, because according to the Equation (5.4) the value of l^{new} could be $l + 1$, $l + 2$ etc. A multiple criticality-level change is a specialization of this case. The case of a multiple criticality-level change is important because tasks of different representative criticality levels need to be suspended in the level-up change.

Error Condition : A new criticality level cannot be determined.

Although Case 1 and Case 2 are inherently covered in the state diagrams and transitions presented above, the error condition is not. Multiple criticality-level changes are special and interesting case to consider. Therefore, we come up with one scenario (Scenario 5) which depicts both the error condition and multiple criticality-level changes.

Now we have identified all the states and transitions relevant to the AMC* scheme. We have also covered the scenarios which are not explicit from the state diagrams. We do not make scenarios for all combinations since some of these are not relevant to the AMC* scheme. However, we cover the scenarios where there is an influence of job/task transitions of one task on another.

5.4 AMC* Scheme: Scenarios

We present the expected system behaviour with the help of six scenarios. In the following six scenarios we cover every transition relevant to the AMC* scheme behaviour at least once. Since an exhaustive case coverage is not possible, the scenarios sufficiently cover the functional aspects (or semantics) of the AMC* Scheme. An overview of the scenarios is presented in Table 5.1. In all six scenarios we assume that $N = 3$ and the system's criticality-level at the start of each scenario equals L^1 . A higher priority value for a task represents a higher priority of the task. Notice that idle time is represented as a task in the graphical representation, as rendered by the Grasp [24] tool.

Table 5.1: Overview of scenarios

Scenarios	Level-up trigger	Task State Transition	Job State Transition	Level-change handler transitions	Level-down trigger
Scenario 1	Level-up trigger (Case 1)	T4, T5, W1	J5 (due to J10 of another job), J8	LH5 (due to LH3), LH6, LH7, LH8	Level-down trigger
Scenario 2 ⁺	Level-up trigger (Case 2)	T3, T5, W1	J6, J8, J9, J10	LH5 (due to LH3), LH6, LH7, LH8	Level-down trigger
Scenario 3 ⁺	Level-up trigger (Case 2)	T5, W1	J2, J4, J7, J8, J9	LH5 (due to LH3), LH7, LH8	Level-down trigger
Scenario 4	Level-up trigger (Case 2)	T3, T5, W1	J8, J9. Note: the previous job is already complete.	LH5 (due to LH3), LH6, LH7	Level-down trigger
Scenario 5 ⁺	- Multiple criticality-level changes (Case 2) - Error Condition	T5, W1	J6, J8, J9	LH5 (due to LH3), LH6, LH8	Level-down trigger
Scenario 6 ⁺	Level-up trigger (Case 1)	T3, T4, T5, W1	J8, J9, J10	LH5 (due to LH8), LH6, LH7	Level-down trigger

⁺ Deactivation of the subsequent releases happens according to the Task Suspend policy (adopted upon a level-up trigger) but the behaviour is not explicit in the scenarios. This is because a level-down trigger occurs before the subsequent periodic release, hence, according to the Task Resume policy (adopted upon a level-down trigger) the tasks are allowed to release new periodic jobs. In other scenarios, the deactivation is visualized.

5.4.1 Scenario 1: Case 1: Task Suspend (including Job Abort) and Task Resume of Ready task

In this section we discuss the criticality level-up trigger that corresponds to Case 1 (a task not completed within its allowed $C_i(L_i)$ WCET). The Job Abort policy for the currently running job is depicted in this scenario (T4, J8 and J10). A Pre-empted job that is allowed to continue execution is also shown in this scenario (J5). A criticality level-down trigger and the Task Resume policy is depicted (T5). Due to criticality level changes, tasks are allocated respective $C_i(l)$ WCETs (W1) which is also covered in this scenario.

The task set is presented in Table 5.2. The expected behaviour is represented in Figure 5.5.

Level-up trigger: Case 1 :

- At time 25ms the current criticality level of the system is $l = L^1$.
- The second job of Task 1 does not complete within its L^1 WCET (J8).
- The new criticality level of the system is $l^{new} = L^2$ (LH6); the new criticality level is identified according to Equation (5.3).
- The currently running job of Task 1 is aborted and Task 1 is suspended (LH7 and LH8 causing J10 and T3, W1).
- Task 3 and Task 2 are allocated $C_i(L^2)$ WCET (W1 for both tasks).
- The pre-empted Task 3 is allowed to continue execution according to its L^2 WCET (J5 due to J10).
- At time 40ms the suspended Task 1 is not allowed to release a new job. Accordingly, the release arrow should not be there; the dashed orange arrow is shown to make the behaviour explicit to the reader.

Level-down trigger :

- At 52ms the job of Task 2 completes its execution (LH3) and an L^2 idle time is identified.
- The system goes back to criticality level L^1 (LH5), which in turn triggers T5 for Task 1. All tasks are allocated corresponding $C_i(L^1)$ WCETs (W1).
- Task 1 releases a new job at $t = 60$ according to Equation (5.1) and Equation (5.2).

Table 5.2: AMC*: Task set 1

Task	Representative Criticality Level	Priority	WCET			Period
			L^1	L^2	L^3	
Task 1	L^1	99	5ms	0	0	20ms
Task 2	L^2	97	18ms	24ms	0	100ms
Task 3	L^3	98	18ms	24ms	24ms	100ms

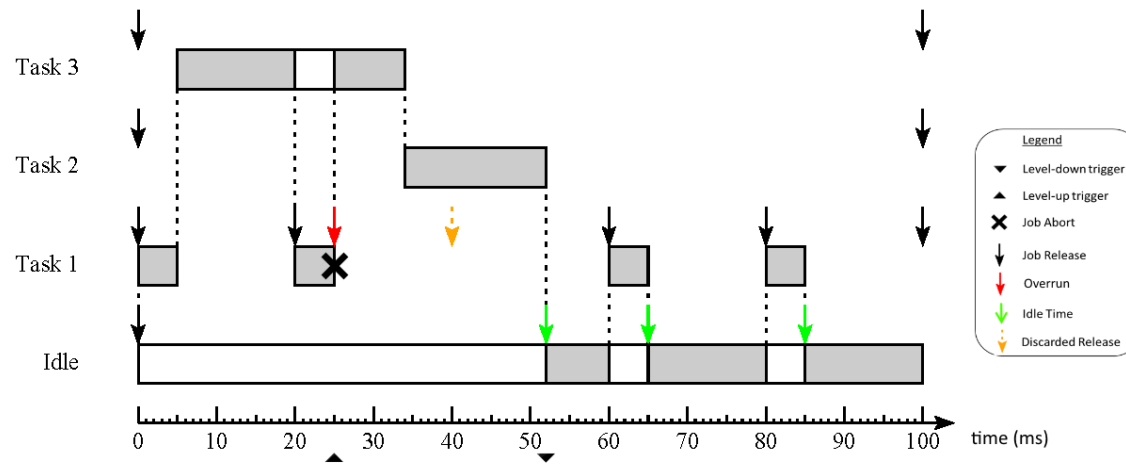


Figure 5.5: AMC* Scheme: Criticality level-up change to L^2 at time 25ms due to a job of Task 1 of representative criticality level L^1 and a level-down change due to a level- L^2 idle time at 52ms.

5.4.2 Scenario 2: Case 2 and Case 1: Job Abort of Contending job

In this section we present two criticality level-up changes. The first criticality level-up trigger corresponds to Case 2 (a task not completed within its WCET at a criticality level lower than its representative criticality level). The currently running job is allowed to continue execution (J9). The second Criticality level-up trigger corresponds to Case 1. The job abort policy for a job in the Contending for CPU state is depicted in this scenario (J6).

The task set is presented in Table 5.3. The expected behaviour is represented in Figure 5.6.

First level-up trigger: Case 2 :

- At time 57ms, the current criticality level of the system is $l = L^1$.
- The second job of Task 2 does not complete within its L^1 WCET (J8).
- The new criticality level of the system is $l^{new} = L^2$ (LH6); the new criticality level is identified according to Equation (5.4).
- Task 1 is suspended (T3).
- Task 2 is allocated $C_2(L^2)$ WCET (W1). Task 1, Task 3 and Task 4 are allocated corresponding $C_i(L^2)$ WCETs (W1).
- The currently running job of Task 2 is allowed to continue execution for $C_2(L^2) - C_2(L^1) = 4\text{ms}$ (LH7 causing J9).

Second level-up trigger: Case 1 :

- At time 61ms, the current criticality level of the system is $l = L^2$.
- The second job of Task 2 does not complete within its L^2 WCET (J8).
- The new criticality level of the system is $l^{new} = L^3$ (LH6), the new criticality-level is identified according to Equation (5.3).
- The currently running job of Task 2 is aborted and Task 2 is suspended (LH7 and LH8 causing J10, T3 and W1).
- A contending job of Task 3 is aborted as well (J6) and the task is suspended (T4, W1).

Level-down trigger :

- At time 73ms, the job of Task 4 completes (LH3) and a L^3 idle time is identified.
- The system goes back to criticality level L^1 (LH5), which in turn triggers T5 for Task 1, Task 2 and Task 3. All tasks are allocated corresponding $C_i(L^1)$ WCETs (W1).
- The release time of each task is identified according to Equation (5.1) and Equation (5.2).

Table 5.3: AMC*: Task set 2

Task	Representative Criticality Level	Priority	WCET			Period
			L^1	L^2	L^3	
Task 1	L^1	99	6ms	0	0	45ms
Task 2	L^2	98	6ms	10ms	0	50ms
Task 3	L^2	97	6ms	6ms	0	50ms
Task 4	L^3	96	6ms	9ms	12ms	60ms

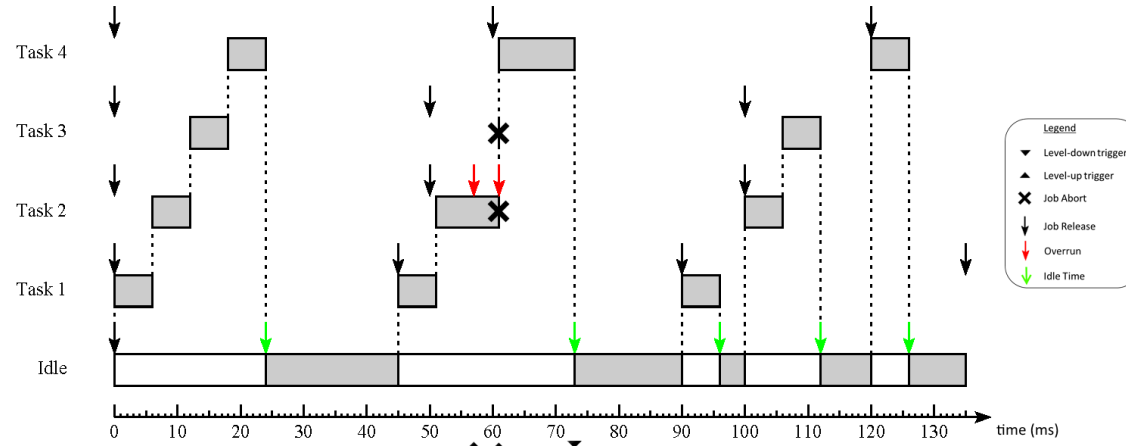


Figure 5.6: AMC* Scheme : Criticality Level-up change to L^2 at time 57ms due to a job of Task 2 and then at time 61ms to L^3 due to a job of Task 2 of representative criticality level L^2 and a level-down change due to a level- L^3 idle time at time 73ms.

5.4.3 Scenario 3: Case 2: Job Abort of Pre-empted Job

In this section we present a scenario with a criticality level-up trigger that corresponds to Case 2 (a task not completed within its WCET at a criticality level lower than its representative criticality level). The abortion of a pre-empted job is depicted in this scenario (J7).

The task set is presented in Table 5.4. The expected behaviour is represented in Figure 5.7.

Level-up trigger: Case 2 :

- A higher priority Task 1 that is allocated the CPU (J2) pre-empts Task 2 (J4) at time 20ms.
- At time 23ms, the current criticality level of the system is $l = L^1$.
- The second job of Task 1 does not complete within its L^1 WCET (J8).
- The new criticality level of the system is $l^{new} = L^2$ (LH6); the new criticality-level is identified according to Equation (5.4).
- Task 1 is allocated $C_1(L^2)$ WCET (W1). Task 2 and Task 3 are also allocated corresponding $C_i(L^2)$ WCETs (W1).
- The currently running job of Task 1 is allowed to continue execution for $C_1(L^2) - C_1(L^1) = 2\text{ms}$ (LH7 causing J9).
- The pre-empted job of Task 2 is aborted (LH8 causing J7).

Level-down trigger :

- At time 25ms, the job of Task 1 completes (LH3) and an L^2 idle time is identified.
- The system goes back to criticality level L^1 (LH5), which in turn triggers T5 for Task 2. All tasks are allocated corresponding $C_i(L^1)$ WCETs (W1).
- The release time of Task 2 is identified according to Equation (5.1) and Equation (5.2).

Table 5.4: AMC*: Task set 3

Task	Representative Criticality Level	Priority	WCET			Period
			L^1	L^2	L^3	
Task 1	L^2	99	3ms	5ms	0	20ms
Task 2	L^1	97	24ms	0	0	100ms
Task 3	L^3	98	9ms	18ms	20ms	100ms

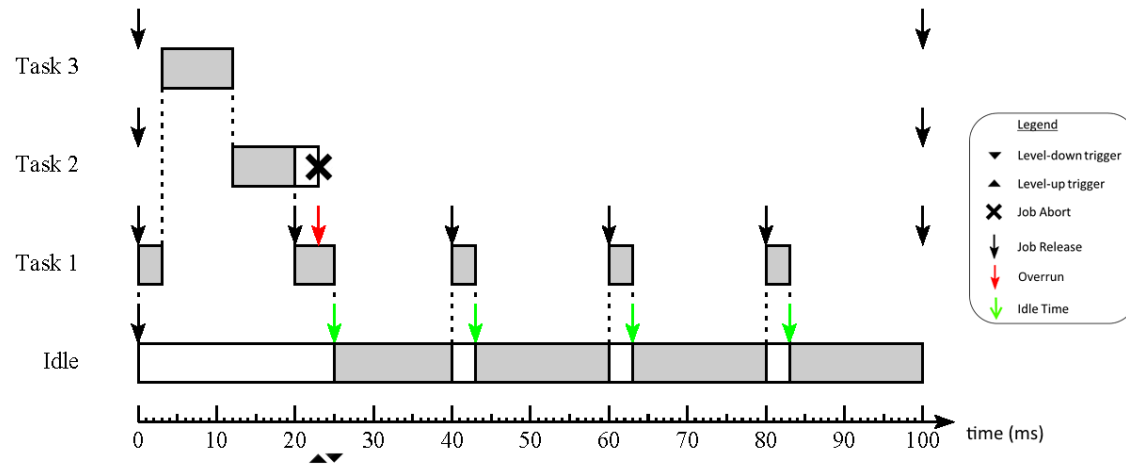


Figure 5.7: AMC* Scheme : Criticality level-up change to L^2 at time 23ms due to a job of Task 1 of representative criticality level L^2 and a level-down change due to level- L^3 idle time at time 25ms.

5.4.4 Scenario 4: Case 2: Task Suspend of Waiting Task and Task Resume

In this section we present a scenario with a criticality level-up trigger that corresponds to Case 2 (a task not completed within its WCET at a criticality level lower than its representative criticality level). Suspension of a Waiting task (T3) and resumption of a task is depicted in this scenario (T5).

The task set is presented in Table 5.5. The expected behaviour is represented in Figure 5.8.

Level-up trigger: Case 2 :

- At time 15ms, the current criticality level of the system is $l = L^1$.
- The first job of Task 3 does not complete within its L^1 WCET (J8).
- The new criticality level of the system is $l^{new} = L^2$ (LH6); the new criticality-level is identified according to Equation (5.4).
- Task 3 is allocated $C_3(L^2)$ WCET (W1).
- Task 3 is allowed to continue execution for $C_3(L^2) - C_3(L^1) = 9\text{ms}$ (LH7 causing J9).
- Task 1 (in the Waiting state) is suspended (T3) hence Task 1 is not allowed to release new jobs at time 20ms and 40ms.

Level-down trigger :

- At time 48ms the job of Task 2 completes (LH3) and an L^2 idle time is identified.
- The system goes back to criticality level L^1 (LH5) which in turn triggers T5 for Task 1. All tasks are allocated corresponding $C_i(L^1)$ WCETs (W1).
- Task 1 releases a new job at 60ms; the release time is calculated according to Equation (5.1) and Equation (5.2).

Table 5.5: AMC*: Task set 4

Task	Representative Criticality Level	Priority	WCET			Period
			L^1	L^2	L^3	
Task 1	L^1	99	6ms	0	0	20ms
Task 2	L^2	97	18ms	24ms	0	100ms
Task 3	L^3	98	9ms	18ms	20ms	100ms

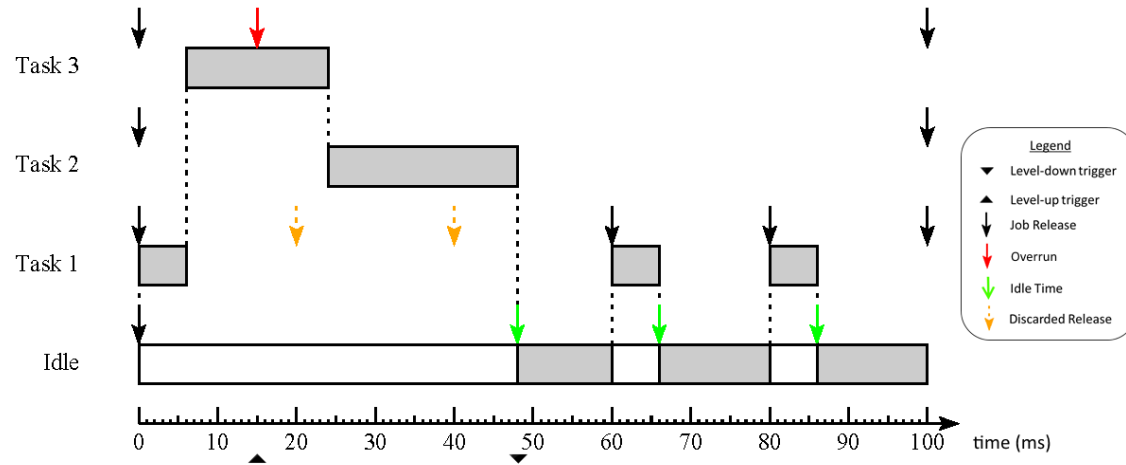


Figure 5.8: AMC* Scheme : Criticality Level-up change to L^2 at time 15ms due to a job of a task of representative criticality level L^2 and a level-down change due to a level- L^2 idle time at time 48ms.

5.4.5 Scenario 5: Case 2: Multiple criticality level change at once and Error Condition

In this section we present a scenario with a criticality level-up change due to Case 2 (a task not completed within its WCET at a lower criticality level than its representative criticality level). This special scenario depicts multiple criticality-level changes at once, accordingly, multiple jobs of tasks with different representative criticality levels are aborted. We also depict the error condition (where l^{new} is not identified). The error condition occurs because the highest criticality tasks miss their deadlines.

The task set is presented in Table 5.6. The expected behaviour is represented in Figure 5.9.

First level-up trigger: Case 2 :

- At time 10ms, the current criticality level of the system is $l = L^1$.
- The first job of Task 3 does not complete within its $C_3(L^1)$ WCET (J8).
- The new criticality level of the system is $l^{new} = L^3$ since $C_3(l) = C_3(L^2) < C_3(L^3)$ (LH6); the new criticality-level is identified according to Equation (5.4).
- Task 3 is allocated $C_3(L^3)$ WCET. Task 1, Task 2 and Task 4 are also allocated corresponding $C_i(L^3)$ WCETs.
- Contending jobs of Task 2 and Task 1 are aborted (J6).
- Task 3 is allowed to continue execution for $C_3(L^3) - C_3(L^1) = 6\text{ms}$ (LH7 causing J9).

Level-down trigger :

- At time 16ms the job of Task 3 completes (LH3) and an L^3 idle time is identified.
- The system goes back to criticality level L^1 (LH5) which in turn triggers T5 for Task 2 and Task 1.
- The previously suspended Task 2 and Task 1 are allowed to release new jobs. All tasks are allocated corresponding $C_i(L^1)$ WCETs (W1).
- Both release new jobs at 30ms; the release time is calculated according to Equation (5.1) and Equation (5.2).

Second level-up trigger :

- At time 34ms, the current criticality level is $l = L^1$.
- Task 4 does not complete in L^1 WCET.
- The new criticality level of the system is $l^{new} = L^2$ since $C_4(l) < C_4(L^2)$ (LH6); the new criticality-level is identified according to Equation (5.4).
- The contending job of Task 1 is aborted and Task 1 is suspended.

Error Condition :

- At time 36ms, the current criticality level of the system is $l = L^2$.
- The second job of Task 4 does not complete within its $C_4(L^2) = C_4(L^3)$ WCET. l^{new} is undefined.

Table 5.6: AMC*: Task set 5

Task	Representative Criticality Level	Priority	WCET			Period
			L^1	L^2	L^3	
Task 1	L^1	96	3ms	0	0	30ms
Task 2	L^2	97	3ms	6ms	0	30ms
Task 3	L^3	98	6ms	6ms	12ms	30ms
Task 4	L^3	99	4ms	6ms	6ms	30ms

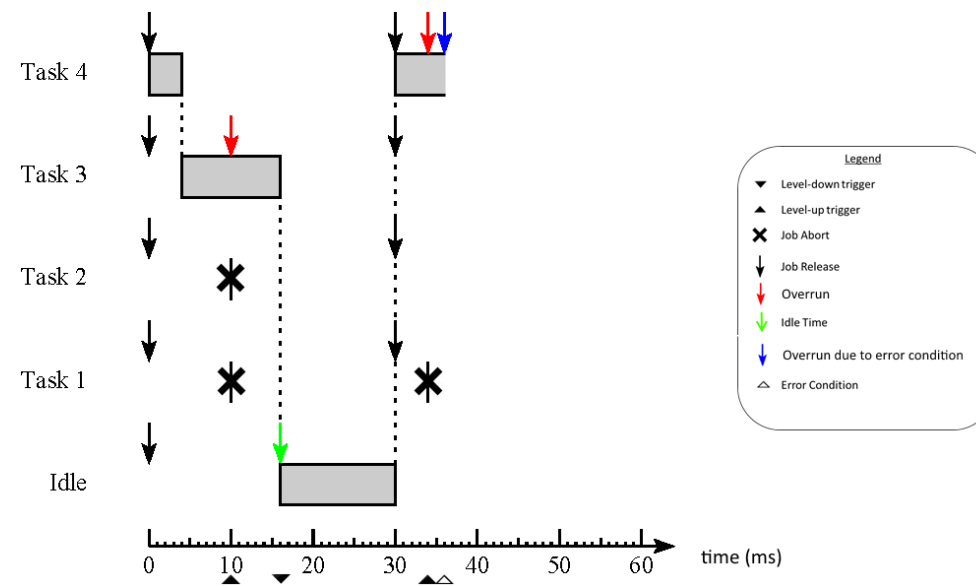


Figure 5.9: AMC* Scheme : First a criticality level-up (multiple) change to L^3 at time 10ms due to a job of a task of representative criticality level L^3 and a level-down change due to a level- L^3 idle time at time 16ms. Then an error condition due to job of a task of representative criticality level L^3 at time 36ms.

5.4.6 Scenario 6: Case 1: Level-down trigger due to a Level-up trigger

In this section we present a scenario with a criticality level-up trigger that corresponds to Case 1 (a task not completed within its WCET at its representative criticality level). This special scenario depicts an immediate criticality level-down trigger due to state transition LH8 (Job Abort/NQ- -) leading to LH5 ($[NQ == 0 \ \&\& \ l > L_i]$ /Level-down) in the state diagram of the Level-change handler (Figure 5.3). This means that when a job is aborted leading to an empty ready queue, the system goes back to the lowest criticality level.

The task set is presented in Table 5.7. The expected behaviour is represented in Figure 5.10.

First level-up trigger: Case 2 :

- At time 41ms, the current criticality level of the system is $l = L^1$.
- The first job of Task 2 does not complete within its L^1 WCET (J8).
- The new criticality level of the system is $l^{new} = L^2$ (LH6); the new criticality-level is identified according to Equation (5.4).
- The currently running job of Task 2 is allowed to continue execution for $C_2(L^2) - C_2(L^1) = 6\text{ms}$ (LH7 causing J9).
- Task 1 is suspended (T3).

Second level-up trigger: Case 1 :

- At time 47ms, the current criticality level of the system is $l = L^2$.
- Task 2 does not complete within its L^2 WCET (J8).
- The new criticality level of the system is $l^{new} = L^3$ (LH6); the new criticality-level is identified according to Equation (5.3).
- Task 2 is allocated $C_2(L^3)$ WCET (W1). Task 1 and Task 3 are also allocated corresponding $C_i(L^3)$ WCETs.
- The currently running job of Task 2 is aborted and Task 2 is suspended (LH7 and LH8 causing T4 and J10).

Level-down trigger :

- At time 47ms, the job of Task 2 is aborted (LH8) and an L^3 idle time is identified.
- The system goes back to criticality level L^1 which in turn triggers T5 for Task 1 and Task 2. Note that the aborted job of Task 2 will not execute further according to the job abort policy. Task 1 and Task 2 are resumed. All tasks are allocated corresponding $C_i(L^1)$ WCETs (W1).
- Task 1 and Task 2 release new jobs at time 50ms and 100ms respectively; the release time is calculated according to Equation (5.1) and Equation (5.2).

Table 5.7: AMC*: Task set 6

Task	Representative Criticality Level	Priority	WCET			Period
			L^1	L^2	L^3	
Task 1	L^1	99	5ms	0	0	50ms
Task 2	L^2	97	18ms	24ms	0	100ms
Task 3	L^3	98	18ms	24ms	24ms	100ms

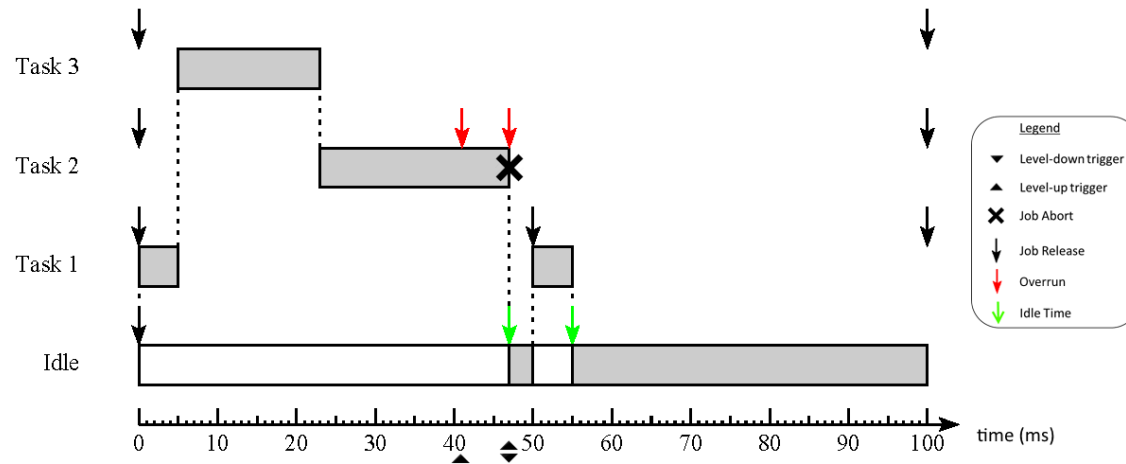


Figure 5.10: AMC* Scheme: Criticality level-up change to L^3 due to a job of a task of representative criticality level L^2 at time 47ms and a level-down change due to a level- L^3 idle time at time 47ms.

Chapter 6

ExSched Framework

In this chapter we discuss the ExSched framework [8] and the generic extensions to the framework. The ExSched framework is a loadable Linux kernel module and is used as the basis for implementing the AMC* scheme. Extensions of the ExSched framework should also be useful for supporting other MC schemes. Selection of ExSched is discussed in Appendix A.

The structural components of ExSched are presented in Figure 6.1. In ExSched, the main ExSched Module resides in kernel space. An application uses the ExSched APIs exposed by the ExSched Library to communicate with the ExSched Module. The communication is relayed to the ExSched Module with the `ioctl()` system call. The ExSched framework supports development of plug-ins with the help of callback functions. Plug-ins for hierarchical scheduling and multi-core scheduling are already provided by ExSched's latest release.

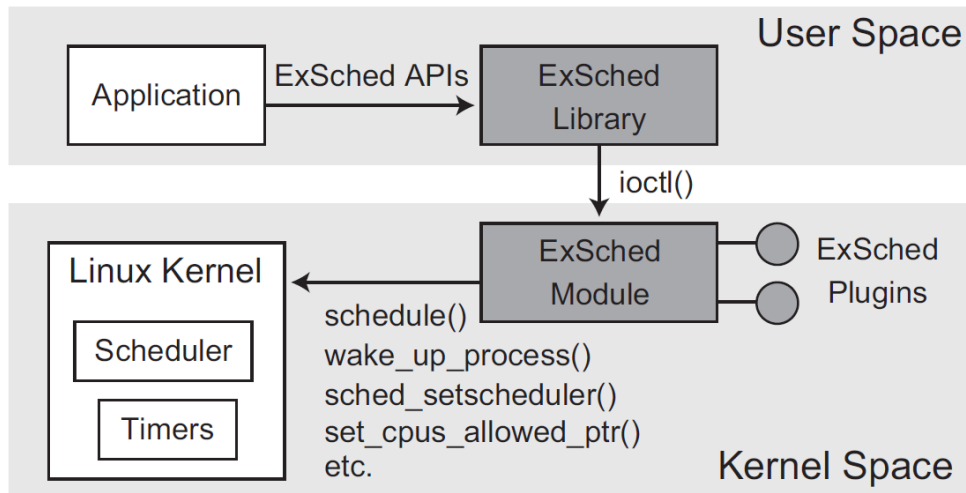


Figure 6.1: ExSched: Structural components [8]

The following subsections provide a detailed overview of the individual components of the ExSched framework.

6.1 ExSched Module

ExSched is an extension of the REal-time SCHEDuler (RESCH) framework by Kato *et al.* [26]. The ExSched Module is a loadable Linux kernel module. It is built as a character-device module. Two configurable scheduling policies for tasks, Earliest Deadline First (EDF) and Fixed-Priority Pre-emptive Scheduling (FPPS),

are provided by the ExSched Module; only one scheduling policy can be used at a time. The ExSched Module relies upon the Scheduler and the Timers provided by the Linux Kernel.

The ExSched Module uses the POSIX-compliant SCHED_FIFO scheduling policy provided by the Linux Kernel. SCHED_FIFO implies that tasks with same priorities are scheduled in FIFO-order. This last property is irrelevant for this project. The priorities of the tasks are managed by the ExSched module according to the configured scheduling policy (EDF or FPPS). The ExSched Module has its own task structure which encapsulates the Linux task structure. ExSched extends the Linux task structure with the timing parameters of the tasks. The ExSched module maintains its own ready queue which contains the task pointers of the tasks in the Ready state. The Linux kernel also maintains its own ready queue (also referred to in literature as the run queue) which maintains a list of all registered tasks. The Linux kernel schedules the tasks based on the state of the tasks defined in the Linux task control block (discussed later in Chapter 8, Section 8.1).

Two important methods which will also be referred to later in this chapter are presented in Table 6.1. These methods are called when the scheduler allocates or deallocates CPU time to a job. The scheduler triggers the Job Start event for a contending job when all higher priority jobs have completed execution. When the Job Start event is handled, the `preempt_curr()` method is called. Upon completion of a job, the Job Complete event is raised. The `preempt_switch()` method is called when the Job Complete event is handled.

Table 6.1: Methods of the ExSched Module which allocate the CPU to the highest priority job selected by the scheduler for execution.

Method	Description
<code>preempt_curr()</code>	This method is called upon a Job Start event triggered by the Scheduler. The method allocates the CPU to the job and pre-empts the currently running job on the CPU, if any.
<code>preempt_switch()</code>	This method is called when a job indicates completion with the Job Complete event. The method allocates the CPU to the highest priority job in ExSched's ready queue, if present.

6.2 ExSched Library

The ExSched Library interacts with the ExSched Module from user space to kernel space using the `ioctl()` system call. The user space API exposed by the ExSched library is used to register/de-register tasks with the ExSched Module and to provide a task's timing parameters like the period of the task. The API exposed by the ExSched Library is presented in Table 6.2. The `rt_init()` (`rt_exit()`) method registers (de-registers) a task with the ExSched module. The `rt_run()` call starts scheduling. A call to `rt_wait_for_period()` causes a job to wait for the next periodic release. The `rt_set_wcet()`, `rt_set_period()`, `rt_set_deadline()` and `rt_set_priority()` methods supply the timing parameters of the task. All these methods are called by the tasks from user space. ExSched Library method calls are relayed to the ExSched Module with the `ioctl()` call. Calls are then mapped within the ExSched Module to calls of ExSched Module methods. This mapping is implemented by a header file (`api.h`) shared between the ExSched Module and the ExSched Library.

Table 6.2: ExSched-API: ExSched Library functions are referred to by a different name in the ExSched Module.

ExSched Library	ExSched Module	Description
<code>rt_init()</code>	<code>api_init()</code>	Change the caller of this function to a real-time task to be scheduled by the ExSched Module.
<code>rt_exit()</code>	<code>api_exit()</code>	Change the caller of this function to a normal task. The task is de-registered from the ExSched Module and runs normally as a Linux task.
<code>rt_run(timeout)</code>	<code>api_run(timeout)</code>	Start ExSched mode in @timeout time. The first job of the task is released when the @timeout expires.
<code>rt_wait_for_period()</code>	<code>api_wait_for_period()</code>	Wait (sleep) until the start of the next period.
<code>rt_set_wcet(wcet)</code>	<code>api_set_wcet(wcet)</code>	Set the worst-case execution time to @wcet.
<code>rt_set_period(period)</code>	<code>api_set_period(period)</code>	Set the minimum inter-arrival time to @period.
<code>rt_set_deadline(deadline)</code>	<code>api_set_deadline(deadline)</code>	Set the relative deadline to @deadline.
<code>rt_set_priority(priority)</code>	<code>api_set_priority(priority)</code>	Set the priority (1-99) to @priority.

6.3 Hierarchical Scheduling Framework

Three HSF plug-ins already have been developed as a part of the ExSched framework namely HSF-FP (Fixed Priority), HSF-EDF (Earliest Deadline First) and Partitioned-HSF-FP. We discuss the HSF-FP plug-in in brief. The HSF-FP plug-in has a data structure defined to store the timing parameters of a server like the budget of the server and its period. For every server a new instance of the structure is created. Each server maintains a list of tasks that belong to the server as an element of the data structure. The HSF-FP plug-in schedules each server based on the timing parameters defined in the structure. A task is allowed to execute only within the budget of its parent server. To prevent a task from executing during the budget of another server, HSF-flags are defined in the ExSched task structure. The flags are inspected and modified by the HSF plug-ins and the ExSched module.

6.4 ExSched: Plug-in development

For plug-in development, it is important to look at the functionality provided by the ExSched Module. Only one plug-in can be installed in ExSched at a time. Installing a plug-in means registering callback functions implemented by the plug-in. The prototypes of the callback functions are declared by the ExSched Module. ExSched plug-ins are invoked by the ExSched Module through the registered callback functions. The ExSched Module exposes four method calls to be utilized by the plug-ins. These four method calls are presented in Table 6.3. When the `api_run()` method is called, the ExSched module notifies the plug-in that the task referred to by the pointer argument of the corresponding callback function will be scheduled by ExSched. When the `api_exit()` method is called, the ExSched module notifies the Linux kernel/scheduler that the task will not be scheduled by ExSched any more. The `api_wait_for_period()` method call notifies the completion of a job to the plug-ins.

The `job_release()` method call notifies the plug-ins about the release of a job. The `job_release()` method, in contrast to the other three API calls, is called only by the Linux kernel upon expiration of the release timer. A sequence diagram representing the calls to the registered callback functions is presented in Figure 6.2. A task is submitted to the ExSched Module for real-time scheduling by a call to the `api_run()` method which triggers the installed plug-in by calling the `task_run_plugin()` method. Notice that in Figure 6.2, the call `api_run()` is not called directly by the application task (see Table 6.2). A task starts to wait for its first periodic release. The `mod_timer()` method sets the Linux timer to expire at the periodic job release of the task. As soon as the timer expires, the `job_release()` method is called and the installed plug-in is invoked by a call to the `job_release_plugin()` method. The first job of the task is scheduled for execution. A task indicates job completion to the ExSched module by calling the `api_wait_next_period()` method. The method invokes the installed plug-in by calling the `job_complete_plugin()` method and then advances the release timer of the task for its next periodic release. A task exits real-time scheduling by calling the `api_exit()` method which invokes the installed plug-in by calling the `task_exit_plugin()` method. Once the installed plug-in is invoked, it may handle the call according to the functionality provided by the plug-in. The difference in the first job and the subsequent jobs of a task is that the first job is scheduled when the task calls the `api_run()` method and the subsequent jobs are scheduled when a task calls the `api_wait_for_period()` method.

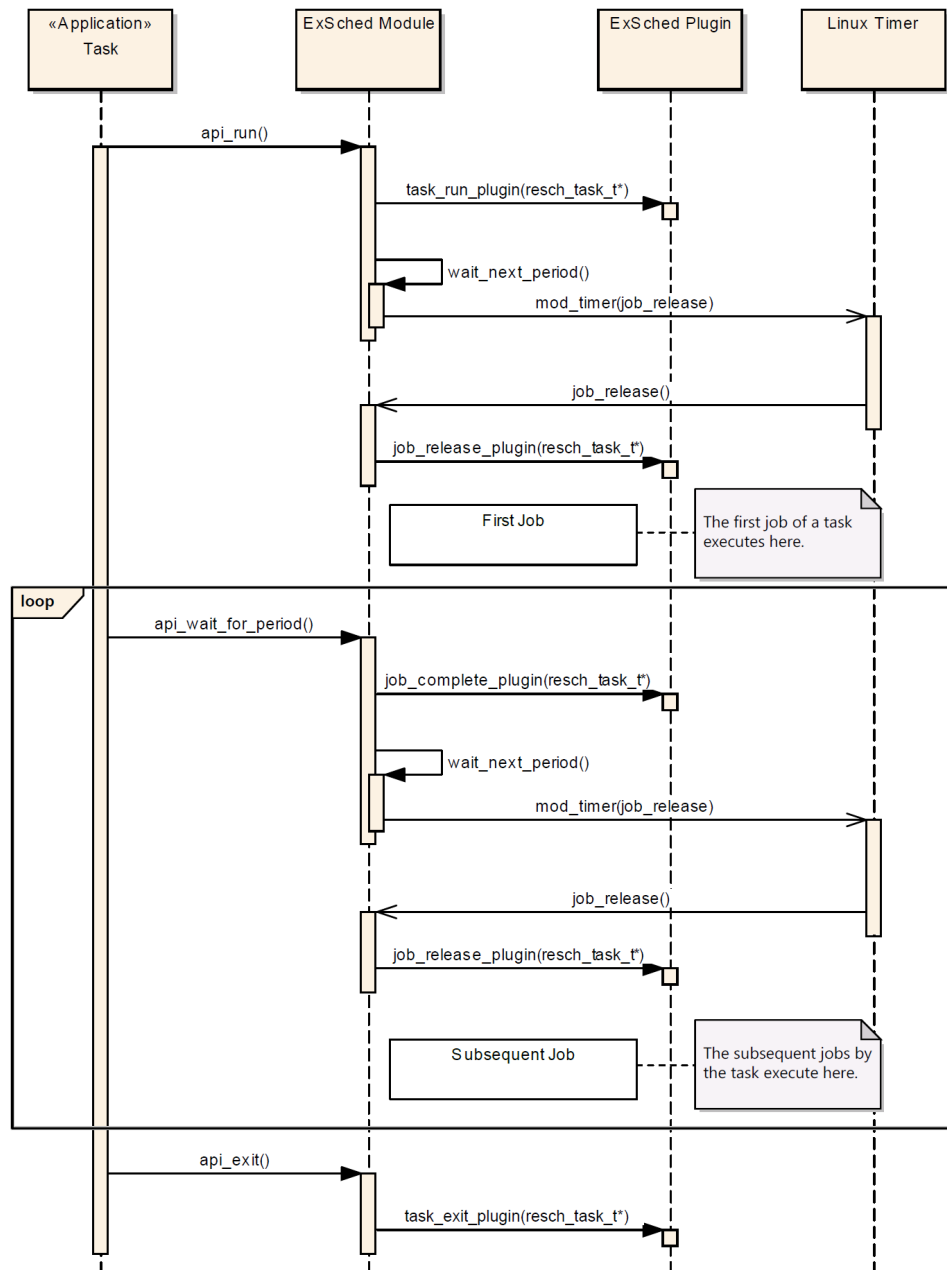


Figure 6.2: ExSched: sequence diagram representing the invocation of the callback functions.

Table 6.3: Method Calls exposed to the ExSched Plug-ins.

Exposed Calls	Method	Interface	Description
api_run()		void (*task_run_plugin) (resch_task_t*)	Called when a task is submitted to ExSched for real-time scheduling using rt_run(timeout). The rt_run(timeout) call is initiated by a task from the user space.
api_exit()		void (*task_exit_plugin) (resch_task_t*)	Called when a task de-registers from ExSched to exit real-time mode using rt_exit(). The rt_exit() call is initiated by a task from the user space.
job_release()		void (*job_release_plugin) (resch_task_t*)	Called when a task releases a new job for execution. The plug-in is called upon expiration of the waiting period.
api_wait_for_period()		void (*job_complete_plugin) (resch_task_t*)	Called when a job is completed. This plug-in method is called when the rt_wait_for_period() call from user space indicates a job completion to the ExSched module. This method is called to start waiting for the next periodic release.

A plug-in is installed and un-installed with the functions listed in Table 6.4. The install_scheduler() call registers the callback functions of the plug-in with the ExSched Module. The uninstall_scheduler() call de-registers the functions of the plug-in.

Table 6.4: API exposed by the ExSched Module to install and un-install a plug-in.

Type	Method
Install Plug-in	<pre>extern void install_scheduler(void (*task_run_plugin)(resch_task_t*), void (*task_exit_plugin)(resch_task_t*), void (*job_release_plugin)(resch_task_t*), void (*job_complete_plugin)(resch_task_t*));</pre>
Un-install Plug-in	<pre>extern void uninstall_scheduler(void);</pre>

6.5 Extending the support for plug-in development

We look at some of the modifications needed to the ExSched Module in order to provide extended functionality for plug-in development. In this section, we discuss the three main modifications made to the ExSched Module. As discussed in Chapter 8, Runtime Monitoring (discussed further in Section 6.5.1) forms the basis for implementing the AMC* scheme. The Task Management services (discussed in Section 6.5.2) like the task suspend-resume functionality and the job abort functionality are needed to realize the criticality-level changes. Detection of idle time (discussed in Section 6.5.3) is required to realize the criticality level-down change. The needed modifications to the ExSched framework to support the AMC* Plugin can be made conform the design of the ExSched framework.

6.5.1 Run-time Monitoring

Run-time monitoring allows a robust control of the system's behaviour in case jobs exceed their WCETs. Run-time monitoring of jobs is essential functionality for supporting Mixed Criticality. We believe that run-time monitoring should be an integral component of the ExSched Module. This is functionality that is useful not just for MC but for other scheduling mechanisms as well. Mechanisms for CPU time reservations are available in the ExSched Module but mechanisms to monitor the WCET of a job are not. Therefore the ExSched Module must be extended to provide support for run-time monitoring. Within the ExSched Module, jobs will be monitored based on the WCET set in the task structure associated with the task.

The timers used in ExSched for raising events like `job_release()` do not satisfy our needs. The values of execution times are stored as 'jiffies', which is the time between two successive clock ticks of the real-time clock. If a task executes for less time than a 'jiffy' before it is pre-empted, the value is rounded up. This makes monitoring imprecise. Therefore we need high resolution timers; the precision of available high resolution timers is in the order of nanoseconds which suits our needs.

The methods introduced in the ExSched Module to handle the monitoring functionality are summarized in Table 6.5. The `start_monitor_timer()` and `stop_monitor_timer()` methods are used to start and stop the monitoring timer of a job. If the timer expires, the `monitor_expire_handler()` method is called. The enumerator `hrtimer_restart` is a return parameter defined in the Linux 'hrtimer.h' file that indicates whether the timer needs to be restarted or advanced to a future moment in time. This is needed for the case when a higher criticality task is running at a lower criticality level with a lower WCET and needs more execution time to complete. So, a criticality level-up change is performed and the timer is advanced with the help of the return parameter i.e., the `hrtimer_restart` enumerator. The `monitor_expire_handler()` method internally invokes a callback function registered by the plug-in.

Table 6.5: Run-time Monitoring. These three methods are local to the ExSched Module and must therefore be implemented in the ExSched Module.

Method	Description
<pre>void start_monitor_timer (resch_task_t *rt)</pre>	<p>This method is called when the job starts execution. When the job starts execution for the first time the timer is started for WCET. If the job was pre-empted and starts execution again, the timer is started for WCET – exec_time. The variable exec_time in the task control block stores the actual execution time of a job cumulatively. Both the variables WCET and exec_time already exist in the ExSched's task control block.</p>
<pre>void stop_monitor_timer (resch_task_t *rt)</pre>	<p>This method is called when a job completes execution or is pre-empted by a higher priority task. When a job is pre-empted, the execution time is added to the exec_time variable.</p>
<pre>enum hrtimer_restart monitor_expire_handler (struct hrtimer *timer)</pre>	<p>This is a timer interrupt handler which is triggered when the monitor timer expires. This method invokes a callback function registered with ExSched which is implemented in the plug-in. The enum hrtimer_restart (defined by the interface of the high resolution timer of the Linux kernel) is an enumerator that denotes whether the timer needs to be restarted. If the return value is HRTIMER_RESTART, the timer will be advanced to WCET – exec_time in our case. If the return value is HRTIMER_NORESTART, the timer will not be advanced.</p>

6.5.2 Task Management Services

To realize criticality-level changes, the functionality to suspend and resume a task is essential. We have introduced services to suspend (`suspend_task()`) and resume (`resume_task()`) a task inside the ExSched Module which can be utilized by the plug-ins (Table 6.6). Task management services, like task suspend and resume, are available in many operating systems like μ C/OS-II [28], FreeRtos [10] and Linux [18].

Along with the suspend-resume functionality, job abortion must be supported by ExSched. The suspend-resume functionality must abort the current job (in state Contending for CPU, Running or Pre-empted; explained in Section 5.2.2) of the task upon suspension. This is accomplished by sending an abort signal to the task by ExSched (so via the Linux Kernel). The actions required upon a criticality level-up change depend upon the task's state (explained in Section 5.2.1):

1. Waiting state, kill the release timer.
2. Ready state, prevent the job from executing further and abort the job.

Next to methods to suspend and resume a task, we introduce a third method which allows the plug-ins to abort a job of an ExSched task. Although exposing this method as part of an API is not necessary for this project because of the abort parameter of `resume_task()`, this is general functionality that is also useful for future plug-ins. In our design, the interface to abort the job is called internally by the ExSched Module. The task pointer is supplied as a parameter to `suspend_task()`, `resume_task()` and `abort_job()` API methods upon criticality-level changes. The parameter indicates to the ExSched Module which task needs to be suspended, resumed, or aborted.

A job is aborted when an idle time is detected. Its task is allowed to continue at that point specifically for this purpose. This is to avoid the overhead of aborting a job while handling the level-up trigger. If we abort a job in the `suspend_task()` method, it will have overhead during the level-up change. If we would decide to flag the job for abortion, we would introduce a new variable in the ExSched task control block and still have the overhead of setting the flag during a criticality level-up change, then inspecting the flag upon level-down change and later maintaining the flag. We choose to avoid this overhead (see Section 4.4) and

abort a job when it is allowed to resume. Note that the `resume_task()` method has a parameter `abort`. If the parameter is set to true the ExSched Module will send the abort signal to the task.

The new API methods introduced in the ExSched Module are listed in Table 6.6.

Table 6.6: API exposing Task Management services to plug-ins. These are functions that must be exposed by the ExSched Module and are declared with the "extern" keyword.

Method	Description
<code>void suspend_task (resch_task_t* rt)</code>	This method moves a task to the Suspended state. If the task was waiting, the method will stop its waiting timer. If the task was ready, the method will dequeue the current job from the ExSched ready queue and inform the Linux kernel not to execute the job.
<code>void resume_task (resch_task_t* rt, bool abort)</code>	This method moves the task from Suspended to either the Waiting or the Ready state if it has a pending job. If the parameter <code>abort</code> is true and the task has a pending job, the task will receive a signal that aborts the pending job.
<code>void abort_job (resch_task_t* rt)</code>	This method allows plug-ins to abort the pending job of an ExSched task.

We look at the design for aborting the job in Chapter 7.

6.5.3 Extended Plug-in Interface

As discussed in the previous chapter, an idle-time event is essential to realize a criticality level-down change. Identifying an idle time is useful in general and not only for this project. For example, some schedulers allow frequency scaling upon idle time; the Linux kernel does the frequency scaling internally, during idle times, using the Linux CPUFreq Governor. Support for idle-time detection helps developers to develop more plug-ins for ExSched in the future. We extend the ExSched Module to provide a hook for an idle-time callback function as ExSched currently does not provide this function. Since ExSched maintains its own ready queue, an idle moment is identified by an empty ready queue. When this is the case, the callback function is invoked. The current interface of the ExSched Module must be modified such that the monitor-expire event and the idle-time event can be utilized by plug-ins. Therefore Table 6.3 must be extended; the extended list of exposed method calls is presented in Table 6.7. When the monitor timer expires, the Linux kernel calls the `monitor_expire_handler()` method of the ExSched Module. The `monitor_expire_handler()` method advances the call to the plug-ins using the registered callback function, i.e., the `monitor_expire_plugin()` method. Similarly, once an idle-time is detected, the registered callback function of the plug-in is invoked.

Table 6.7: Extension to the API for ExSched Plug-ins.

Exposed Method Calls	Interface	Description
<code>monitor_expire_handler()</code>	<pre>void (*monitor_expire_plugin) (resch_task_t*)</pre>	Called when the monitor timer of the job in execution expires before the job completes execution. The <code>monitor_expire_handler()</code> method calls the callback function for monitor expiration registered by the plug-in. After the <code>monitor_expire_plugin()</code> is executed the <code>monitor_expire_handler()</code> method advances the monitoring timer if the job is allowed to continue execution after a criticality level-up change.
-	<pre>void (*idle_time_plugin) (resch_task_t*)</pre>	Called when the ExSched Module identifies an idle time, i.e., when the ExSched's ready queue is empty. An empty ready queue is identified by inspecting the ExSched's ready queue in the ExSched Module. The ready queue is inspected twice, i.e., when a job completes execution and after a criticality level-up change is performed.

The interface to install a plug-in and to pass the callback functions needs to be updated. The updated interface with added parameters for the monitor-expire and idle-time callbacks is presented in Table 6.8.

Table 6.8: Updated API exposed by the ExSched Module to install and un-install a plug-in.

Type	Method
Install Plug-in	<pre> extern void install_scheduler(void (*task_run_plugin)(resch_task_t *), void (*task_exit_plugin)(resch_task_t*), void (*job_release_plugin)(resch_task_t*), void (*job_complete_plugin)(resch_task_t*), void (*monitor_expire_plugin)(resch_task_t*), /*called when a job is not complete within its allowed WCET*/ void (*idle_time_plugin)(resch_task_t*) /*called when the ExSched module identifies idle time*/); </pre>
Un-install Plug-in	<pre> extern void uninstall_scheduler(void); </pre>

Chapter 7

System Design

In this chapter, we look at a design which will lead towards an implementation of a multiple criticality level scheduler. The chapter presents the design for the AMC* scheduler. The design consists of an ExSched plug-in for AMC* specific functionality and its interfaces to the ExSched Module in kernel space and to tasks in user space. The design is for a single-core processor running Linux.

7.1 Design Influencers

The main criterion for designing an MCS is minimizing the computation time while handling a criticality level-up trigger and if possible postponing the overheads to a later moment. For example, if job abortion can be delayed/postponed to a later moment in time, say an idle time, it will significantly reduce system overheads upon a criticality level-up trigger. Hence, system performance upon occurrence of a criticality level-up trigger is the key criterion for an MCS design.

7.2 AMC* Plugin Module structure

The structure of the ExSched framework allows a plug-in based design. We have decided to make a plug-in for implementing the AMC* scheme, which will use the extended interface to support plug-in development. We use a similar structure for AMC* as for ExSched itself, i.e., a library and a kernel module. Addition of the AMC* components do not actually modify the ExSched framework. Figure 7.1 shows the structure of the ExSched framework with the AMC* plug-in. Figure 7.1 is an extension of Figure 6.1. The components introduced in the figure are the AMC* Plugin Module, the AMC* Library, the Plug-in Support API and the Callback functions. The `ioctl()` between the ExSched Library and the ExSched Module is the same as in Figure 6.1. The `ioctl()` (AMC*) interface enables a user space task to communicate with the kernel space AMC* Plugin Module. The Install Plug-in API is the extended API discussed earlier in Section 6.5.3.

The AMC* API exposed by the AMC* Library is used to set the representative criticality level and the WCET for each criticality level in the AMC* Plugin Module. The AMC* Plugin Module and the AMC* Library handle the AMC*-scheme related functionality. The Plug-in Support API is a general purpose API to be utilized by any plug-in, not only the AMC* plug-in. The Callback functions interface are handlers to the various events triggered by the ExSched module.

The functionality of the new components in Figure 7.1 is the following.

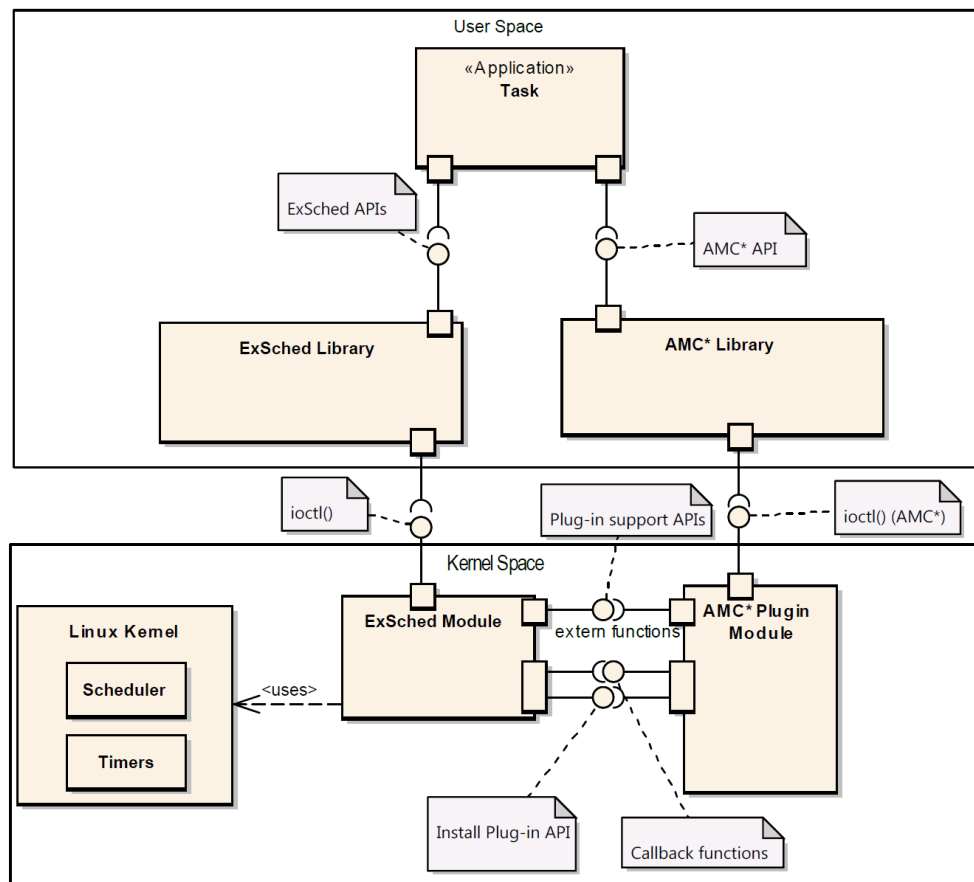


Figure 7.1: Static structure extended with the AMC* Plugin Module.

AMC* Plugin Module : This AMC* Plugin module is developed to handle actions like Task Suspend and Task Resume needed to realize the AMC* scheme. The AMC* Plugin Module defines the number of criticality levels supported. This module stores a pointer to each task submitted to the ExSched Module. This module also stores the representative criticality level of each task and the WCETs of the task per criticality level and stores and maintains the current criticality level of the system. This module is a loadable Linux kernel module which relies on the ExSched Module for the real-time scheduling of tasks.

AMC* Library : The AMC* library is positioned similarly to the ExSched Library in user space. The AMC* Library interacts with the AMC* Plugin Module from user space to kernel space using the `ioctl()` system call (represented by the label '`ioctl() (AMC*)`' in Figure 7.1), similar to the ExSched Library. The method of communication using the `ioctl()` call hides all implementation details of the kernel space AMC* Plugin Module from user space.

Tasks must be aware of the number of criticality levels supported by the system. The AMC* Library must expose an API method which allows tasks to specify their representative criticality level and their WCET per criticality level. The API of the AMC* Library is given in Table 7.1.

Table 7.1: AMC* Library.

Method	Description
NO_OF_CRIT_LEVELS	This is a constant defined in a shared header file (between the AMC* Library and the AMC* Module) which denotes the number of criticality levels supported by the system. The value is statically assigned while configuring the system before initialization.
int rt_set_rep_crit_level(int rep_crit, unsigned long[NO_OF_CRIT_LEVELS] wcet_per_crit)	This API call allows a task to indicate its representative criticality level and its WCET at each criticality level. The first parameter is the representative criticality level and the second parameter is an array indicating the WCET at each criticality level. Note that the rt_set_wcet() method in Table 6.2 will be used but the WCET of a task will be allocated again by the AMC* plug-in. When the rt_set_rep_crit_level() method is called the AMC* plug-in allocates a new WCET based on the current criticality level of the system.

Plug-in Support API : ExSched provides this API which is required by the AMC* Plugin Module to realize the AMC* scheme. The API extended with the functionality explained in Section 6.5.3 provides the task management services. This API has been introduced in Section 6.5.2 (Table 6.6). While performing a criticality level change, the AMC* plug-in needs to indicate to the ExSched module which tasks need to be suspended or resumed. To communicate with the ExSched module, the pointers to all ExSched tasks should be available to the AMC* plug-in. The AMC* plug-in also needs to store the representative criticality level and the WCET per criticality-level for each task. These properties should be associated with the corresponding ExSched pointers which are not available in the AMC* plug in. When a task provides its AMC* properties at initialization, its ExSched pointer can be made available to the AMC* plug in by adding an ExSched API call get_current_task() that returns the ExSched pointer. The extended API is presented in Table 7.2.

Table 7.2: Extended API exposing Task Management services to plug-ins.

Method	Description
resch_task_t* get_current_task()	This method allows plug-ins to get the ExSched task pointer of the currently running task.

Callback functions : Callback functions implemented within the AMC* Plugin Module are calls invoked by the ExSched Module. The callback functions for the monitor-expire and idle-time events implemented within the AMC* Plugin Module are listed as parameters of the install_plugin() method in Table 6.8 and are listed again in Table 7.3.

Table 7.3: Callback functions implemented in the AMC* Plugin Module.

Method	Description
void monitor_expire_plugin_handler (resch_task_t *rt)	This callback function is invoked by the ExSched Module if a job of a task does not complete within its allocated WCET.
void idle_time_plugin_handler (resch_task_t *rt)	This callback function is invoked if idle time is identified by the ExSched Module.

7.3 State Diagrams

We present three state diagrams namely the Task State Diagram, the Job State Diagram and the Level Handler state diagram. The implementation of the Task State Diagram resides in the ExSched Module which manages these states. The implementation of the Job State Diagram resides in the ExSched Module which uses the functionality provided by the Linux Kernel to manage the job states. The Level Handler is implemented in the AMC* Plugin. The states of the Level Handler are managed by the AMC* Plugin and uses the task management services provided by the ExSched module.

In the state diagrams, two methods `hold_job()` and `unhold_job()` appear. These methods are symbolic representations. The `hold_job()` method represents that the job of a task is not allowed to execute further. The `unhold_job()` method represents that the job of a task is allowed to contend for CPU time again. To keep the state diagrams simple, we do not model the scheduler. Therefore, in this section, we use symbolic representations of the `set_tsk_need_resched()` and `clear_tsk_need_resched()` methods provided by the Linux scheduler (details of these methods are given in the Implementation chapter). In the state diagrams, `allocate_WCET()` is also a symbolic representation of allocating the $C_i(l)$ WCET to a task which is performed by the AMC* plug-in upon criticality-level changes.

7.3.1 Task State Diagram

The execution states of a task in ExSched and the change in the WCET of a task are presented in Figure 7.2. The state diagram presented here is similar to the task state diagram (Figure 5.1) presented in the semantics chapter. More details and ExSched-specific function names have been added in this section.

There are three execution states in the Task State Diagram namely Waiting, Ready and Suspended. We have introduced the Suspended state as a new state in the ExSched Module. Upon occurrence of a level-up trigger, tasks with a representative criticality lower than the new criticality level move from the Waiting or Ready state to the Suspended state. The release timers of tasks that are suspended while in the Waiting state are stopped and the task moves to the Suspended state. The tasks that are suspended while in the Ready state are not allowed to contend for the CPU any more and are moved to the Suspended state. Note that the released jobs of lower criticality tasks are aborted only upon occurrence of a level-down trigger (see Section 6.5.2). A job has to abort itself (discussed in next section), so the tasks are moved from the Suspended state back to the Ready state for this purpose.

The task property that changes during runtime due to criticality level changes is its WCET. We model this change of WCET as a forked state in the state diagram presented in Figure 7.2.

Task States : The following are the states of an ExSched Task.

Standard task states : The following are the three standard states of an ExSched task.

Waiting : A task enters this state upon initialization. The task waits for its first release by setting a release timer. When a task enters the Waiting state, the "waiting" flag in its task control block is set to true.

Ready : A task enters this state when the release timer expires and the task releases a new job. When a task enters this state, its "waiting" flag in the TCB is set to false. The released job is queued in the ready queue for execution based on its priority. When a task moves out of this state, its job is removed from the ready queue.

WCET : This state corresponds to the allocation of execution time in standard ExSched. As soon as a task initializes, it enters this forked state. For the AMC*-related functionality, the task is allocated a WCET corresponding to the system's current criticality level.

Added States : The following is a state of a task introduced in the ExSched Module to realize MCS. If the ExSched framework would have inherently supported the suspend-resume functionality this state would already be a part of the framework.

Suspended : A task can be suspended from either of the Waiting and Ready states. If a task was in the Waiting state before being suspended it will not be allowed to release a new job. If a task was in the Ready state before being suspended its running or pending jobs are not allowed to continue execution and become holding jobs.

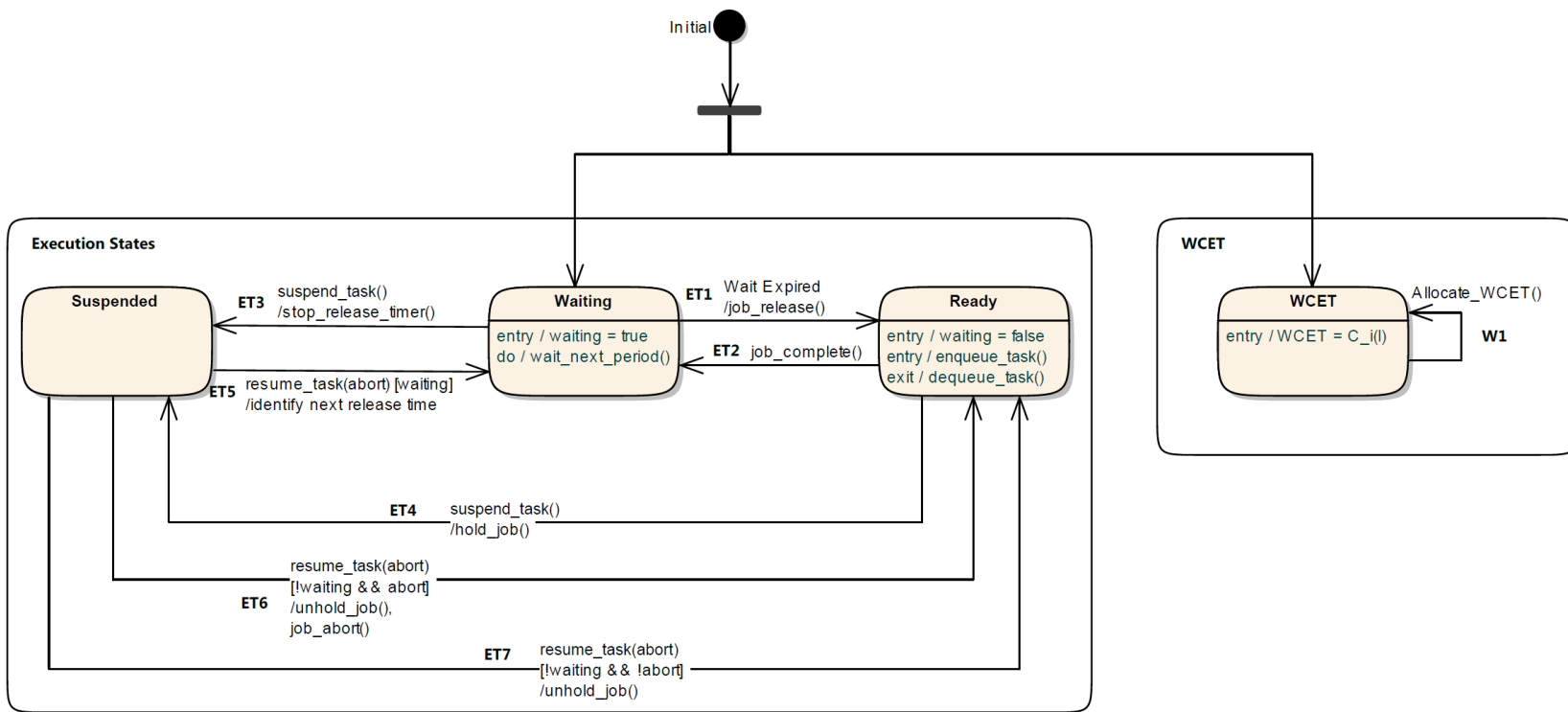


Figure 7.2: ExSched: task state diagram.

Task state transitions : The following are the task state transitions.

Standard task transitions :

- ET1** Waiting to Ready : `Wait Expired /job.release()` → This transition occurs when the release timer for the task expires and the task releases a new job.
- ET2** Ready to Waiting : `job.complete()` → This transition occurs when the job signals completion.

Added task transitions :

- ET3** Waiting to Suspended : `suspend_task() /stop_release_timer()` → This transition occurs when a task is suspended while it is waiting for next release. The release timer of the task is stopped. Consequently, the task will not release a new job as long as it is in the Suspended state.
- ET4** Ready to Suspended : `suspend_task() /hold_job()` → This transition occurs when a task is suspended while it is in the Ready state, i.e., the task has a running or pending job. The task is suspended and it is indicated to the Linux scheduler that the task should not be scheduled unless indicated otherwise. This is achieved by calling the `hold_job()` method.
- ET5** Suspended to Waiting : `resume_task(abort) [waiting]` /identify next release time → When a task can be resumed, the `waiting` flag in the TCB is checked; if the flag is true then this transition occurs. A new release time is determined. Note that there is no pending job of the task which needs to be aborted since either the jobs have completed execution or no job was released yet. Hence, the `abort` parameter is not significant here.
- ET6** Suspended to Ready : `resume_task(abort) [!waiting && abort]` /`unhold_job()`, `job_abort()` → When a task can be resumed, the `waiting` flag in the TCB is checked. If the flag is false then this transition occurs. The job of the task is allowed to contend for CPU time. If the parameter `abort` in the `resume_task()` API call was true then `job_abort()` is triggered. This results in the job aborting itself when it is scheduled for execution again.
- ET7** Suspended to Ready : `resume_task(abort) [!waiting && !abort]` /`unhold_job()` → The difference from the transition ET6 is that if the parameter `abort` in the `resume_task()` API call was false, the `job_abort()` is NOT triggered. Therefore, the job is allowed to continue execution. For our design this transition is not relevant since the pending jobs of all lower criticality tasks must be aborted; the `abort` parameter will always be true.
- W1** WCET to WCET : `allocate.WCET()` → This transition occurs when a new WCET must be allocated to the task. The variable WCET in the TCB is assigned a new value. After a new WCET is set for the task, the new WCET value is inspected by the `monitor_expire_handler()` method and the monitoring timer is advanced accordingly (see Section 6.5.1).

In our approach the method `resume_task()` has a parameter to abort a job (instead of `suspend_task()`); if this parameter is true, the ExSched Module will simply check the `waiting` flag. If the `waiting` flag is false and the `abort` parameter is true, the pending job is aborted. This approach is simple and all overheads (setting and inspecting the flags) are postponed to an idle time (i.e., a level-down trigger). Although the approach is not intuitive at the first glance, it avoids overheads during a criticality level-up change.

7.3.2 Job State Diagram

The job state diagram is presented in Figure 7.3. The job state diagram differs from the one presented in the semantics chapter. We have added more details and two states are added in the state diagram in order to achieve the Job Abort; the state transitions differ accordingly. The Vanilla (also referred to as Mainline) versions of the Linux kernel are not inherently real-time. As a consequence, the mainline versions of the Linux kernel do not have a notion of a job. Therefore, no means are provided by the kernel to abort a particular job of a task. However, it is possible to defer job execution. We use the symbolic representation `hold_job()` to represent that a job is not allowed to continue execution. We added the Holding state to represent the state where the job is neither running nor contending for the CPU. We use the symbolic notation `unhold_job()` to represent that the job is allowed to continue execution. As Linux does not provide means to abort a particular job, this required introduction of the Abort Job state within the Running state. When a level-down trigger occurs, all the released jobs of the lower criticality tasks (in the Holding state) are aborted.

Job States : The following are the states of a job.

Standard job states :

Idle : State before a job is released by a task.

Contending for CPU : A released job enters this state and (indirectly) calls the Linux Scheduler method `schedule()`. It then waits until the scheduler has executed all higher priority jobs. After all jobs with higher priority are executed, the `schedule()` method returns control to the contending job for execution.

Running : A job enters this state when the CPU is allocated to it. There are two sub states within the Running state, one of them is an added state. The standard sub state is:

Execute Job : When a job is in this state it continues execution until completion. When a job enters this state, the monitor timer is started for its $WCET - exec_time$. The variable `exec_time` keeps track of the time that a job has executed. When the job starts, `exec_time = 0`. If the job is pre-empted, the time that it executed is added to `exec_time` which thus represents the total amount of CPU time a job has consumed. If the job is resumed, the time set for the monitor equals $WCET - exec_time$ where `exec_time` ≥ 0 . When a job exits this state, the monitor timer is stopped and the execution time of the job is cumulatively saved in `exec_time`. The action `cumulatively_save_exec_time()` is a symbolic representation of the functionality to store the total amount of CPU time consumed by a job.

Pre-empted : A job enters this state if a higher priority job arrives and the scheduler allocates the CPU to that job.

Added states :

Running : The added sub state is:

Abort Job : When a job enters this state it is aborted. The job abort handler is executed after which the job aborts itself.

Holding : This is the state of a job when it is not allowed to continue execution. A job enters this state when its task is suspended and the method `hold_job()` is called.

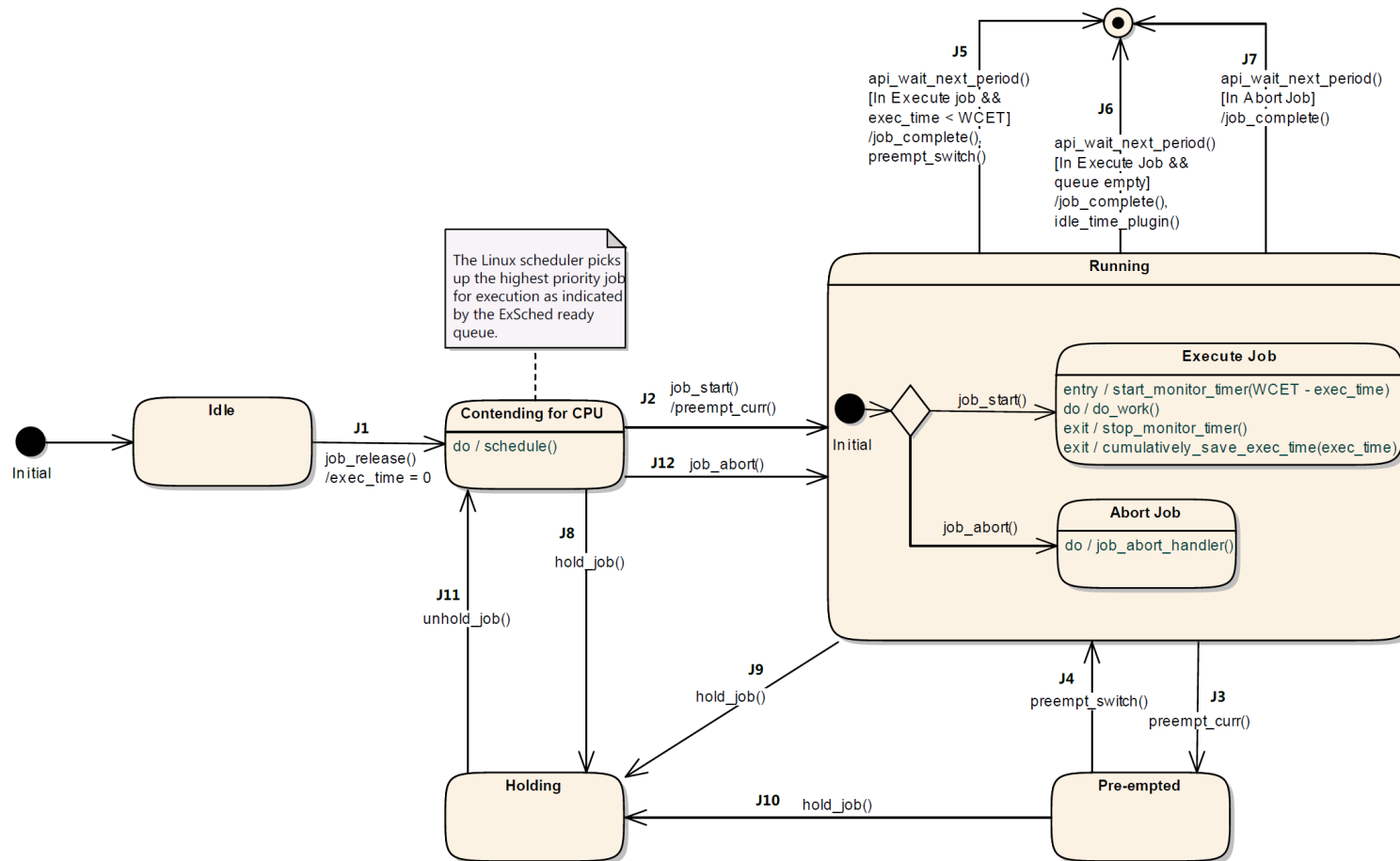


Figure 7.3: Job state diagram.

Job state transitions : The following are the job state transitions.

Standard job state transitions :

- J1** Idle to Contending for CPU : `job_release()` / `exec_time = 0` → This transition occurs when a task releases a new job. The `exec_time` of the job is set to 0.
- J2** Contending for CPU to Running : `job_start()` / `preempt_curr()` → The arrival of a contending job with higher priority than the current job causes this transition. The higher priority job starts execution. The lower priority job in the Running state is pre-empted (`preempt_curr()`).
- J3** Running to Pre-empted : `preempt_curr()` → When a lower-priority job is in the Running state and a higher-priority job is released, the scheduler allocates the CPU to the higher-priority job. The lower-priority job is pre-empted and this transition occurs. This transition is triggered by a call to `preempt_curr()`.
- J4** Pre-empted to Running : `preempt_switch()` → This transition occurs when the CPU is allocated to a pre-empted job by the scheduler. This transition is triggered by a call to `preempt_switch()`.
- J5** Running to Final : `api_wait_next_period()` [In Execute Job && `exec_time < WCET`] / `job_complete()`, `preempt_switch()` → This transition occurs when a job actually finishes execution within its WCET. The sub state of the job must be Execute Job. This transition is triggered by the `api_wait_next_period()` call (this is same as `rt_wait_next_period()` in the ExSched library). A `job_complete()` event is raised and a highest priority job in the ExSched's ready queue is allowed to continue execution.

Added job state transitions :

- J6** Running to Final : `api_wait_next_period()` [In Execute Job && `queue_empty`] / `job_complete()`, `idle_time_plugin()` → This transition occurs when the job completes execution and there are no more jobs in the ready queue, i.e., the ready queue is empty. The sub state of the job must be Execute Job. This transition is triggered by the `api_wait_next_period()` call. A `job_complete()` event is raised and `idle_time_plugin()` is invoked.
- J7** Running to Final : `api_wait_next_period()` [in Abort Job] / `job_complete()` → This transition occurs after the job abort signal is handled. The sub state of the job must be Abort Job. This transition is triggered by the `api_wait_next_period()` call. A `job_complete()` event is raised which notifies the scheduler to set the next release time.
- J8** Contending for CPU to Holding : `hold_job()` → This transition occurs when a contending job of a lower-criticality task is not allowed to execute further due to a criticality level-up change. This transition is triggered by a call to the `hold_job()` method.
- J9** Running to Holding: `hold_job()` → This transition occurs when the Running job (in the sub state Execute Job) is not allowed to execute further. This transition is triggered by a call to the `hold_job()` method.
- J10** Pre-empted to Holding : `hold_job()` → This transition occurs when a pre-empted job is not allowed to execute further. This transition is triggered by a call to the `hold_job()` method.
- J11** Holding to Contending for CPU : `unhold_job()` → This transition occurs when the Holding job is allowed to continue execution. This transition is triggered by a call to the `unhold_job()` method.
- J12** Contending for CPU to Running : `job_abort()` → This transition occurs when a job must be aborted. The job goes to the Running state and the sub state of the job is Abort Job, where the job is aborted by the `job_abort_handler()` method. Note that this transition occurs after an idle-time is detected.

Although the transition J4 is a standard transition, it becomes relevant for the AMC* scheme when it is a consequence of the transition J9. This is similar to the J5 transition discussed in the previous chapter.

7.3.3 Level Handler State Diagram

The Level Handler state diagram presented here also differs from the one in Chapter 5. In our design everything related to the AMC* aspects of the tasks and criticality levels is maintained by the level handler. The criticality-level changes are handled by the AMC* plug-in and the ready queue is inspected and maintained by the ExSched module. Therefore, we introduce states of the Level Handler which represent the AMC* functionality only. In this state diagram, the event `monitor_expire_plugin()` is initiated by the ExSched module when a job does not complete within its $C_i(l)$ WCET where l is the current criticality level of the system. Likewise, the `idle_time_plugin()` is initiated by the ExSched module when the ready queue is empty. The ready queue is inspected either when a job is completed or after a level-up change is performed (see Section 5.4.6).

In the Level Handler state diagram there are five states namely Idle, Overrun, Level-Up Change, Idle Time and Level-Down Change. Upon occurrence of a Level-Up trigger, the new criticality level of the system is determined in the Overrun state. Once the new criticality level of the system is determined, the tasks are handled in the Level-Up Change state. Similarly when an idle time occurs, the current criticality level of the system is checked. If the current criticality level of the system is not the lowest criticality level, a level-down change is performed and all tasks are handled in the Level-Down Change state.

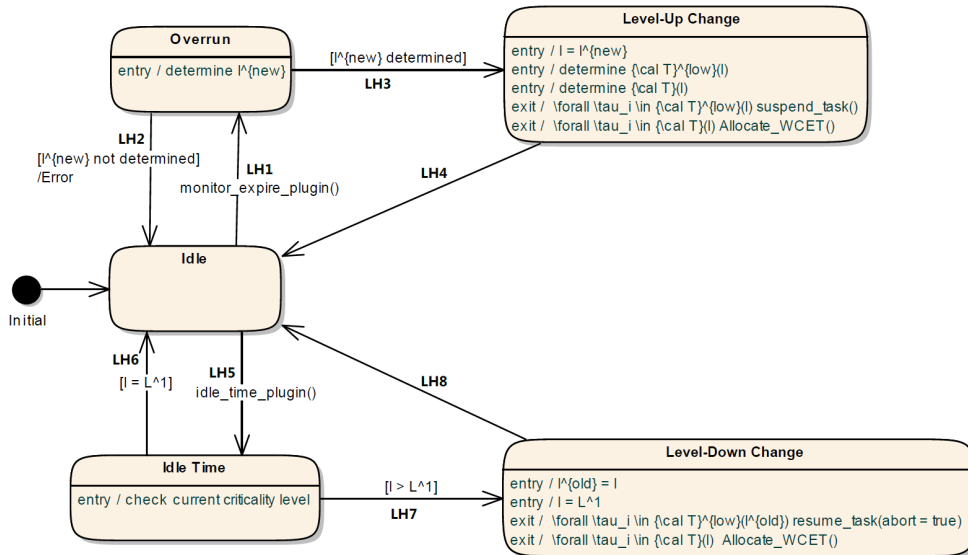


Figure 7.4: Level Handler state diagram. We have used the Latex [4] notations here.

Level Handler States : The AMC* plug-in which handles the criticality level changes has the following five states:

Idle : This is the state of the plug-in when tasks are executing normally and no job exceeds its WCET at the current criticality level of the system.

Overrun : This is the state of the plug-in when a job exceeds its $C_i(l)$ WCET. As soon as the plug-in enters this state, a new criticality level of the system is determined.

Level-Up Change : This is the state of the plug-in when a level-up change is being executed. The system is assigned the new criticality level determined in the Overrun state. Then the two sets $\mathcal{T}^{low}(l)$ ($\{\text{cal } T\}^{low}(l)$) and $\mathcal{T}(l)$ ($\{\text{cal } T\}(l)$) are determined.

Idle Time : This is the state of the plug-in when an idle time is identified. As soon as the plug-in enters this state, the current criticality level of the system is inspected.

Level-Down Change : This is the state of the plug-in when a criticality down change is being executed. In this state, first the old criticality level is stored and then the lowest criticality level L^1 is set as the current criticality level.

Level Handler state transitions : The following are the state transitions:

LH1 Idle to Overrun : `monitor_expire_plugin()` → This transition occurs when a job is not finished within its $C_i(l)$ WCET. The monitor expires and the plug-in is invoked using the `monitor_expire_plugin()` call.

LH2 Overrun to Idle : $[l^{new} \text{ not determined}] / \text{Error}$ → This transition occurs when a new criticality level of the system is not determined, i.e., the error condition occurs. We assume this condition does not occur.

LH3 Overrun to Level-Up Change : $[l^{new} \text{ determined}]$ → This transition occurs when a new criticality level l^{new} of the system is determined.

LH4 Level-Up Change to Idle → This transition occurs when the current criticality level of the system is changed to the new criticality level. When this transition occurs, the exit actions of the Level-Up Change state are performed. All tasks in $\mathcal{T}^{low}(l)$ are suspended ($\forall \tau_i \in \{\text{cal } T\}^{low}(l) \text{ suspend_task}()$) and all tasks in $\mathcal{T}(l)$ are allocated respective $C_i(l)$ WCETs ($\forall \tau_i \in \{\text{cal } T\}(l) \text{ allocate_WCET}()$).

LH5 Idle to Idle Time : `idle_time_plugin()` → This transition occurs when an idle moment is identified by the ExSched Module and the plug-in is invoked using the `idle_time_plugin()` call.

LH6 Idle Time to Idle : $[l = L^1]$ → This transition occurs when the current criticality level of the system was already the lowest criticality level.

LH7 Idle Time to Level-Down Change : $[l > L^1]$ → This transition occurs when the system's current criticality level is higher than the lowest criticality level.

LH8 Level-Down Change to Idle → This transition occurs when the current criticality level of the system is changed to the lowest criticality level. When this transition occurs the exit actions of the Level-Down Change state are performed. All tasks in $\mathcal{T}^{low}(l^{old})$ are resumed ($\forall \tau_i \in \{\text{cal } T\}^{low}(l^{old}) \text{ resume_task}(\text{abort} = \text{true})$). Since $l = L^1$ all tasks in $\mathcal{T}(l)$ are allocated their respective $C_i(L^1)$ WCETs ($\forall \tau_i \in \{\text{cal } T\}(l) \text{ allocate_WCET}()$).

7.4 Sequence Diagrams

The state changes are further illustrated with the help of the following two sequence diagrams.

Level-up change : A criticality level-up change is presented in Figure 7.5.

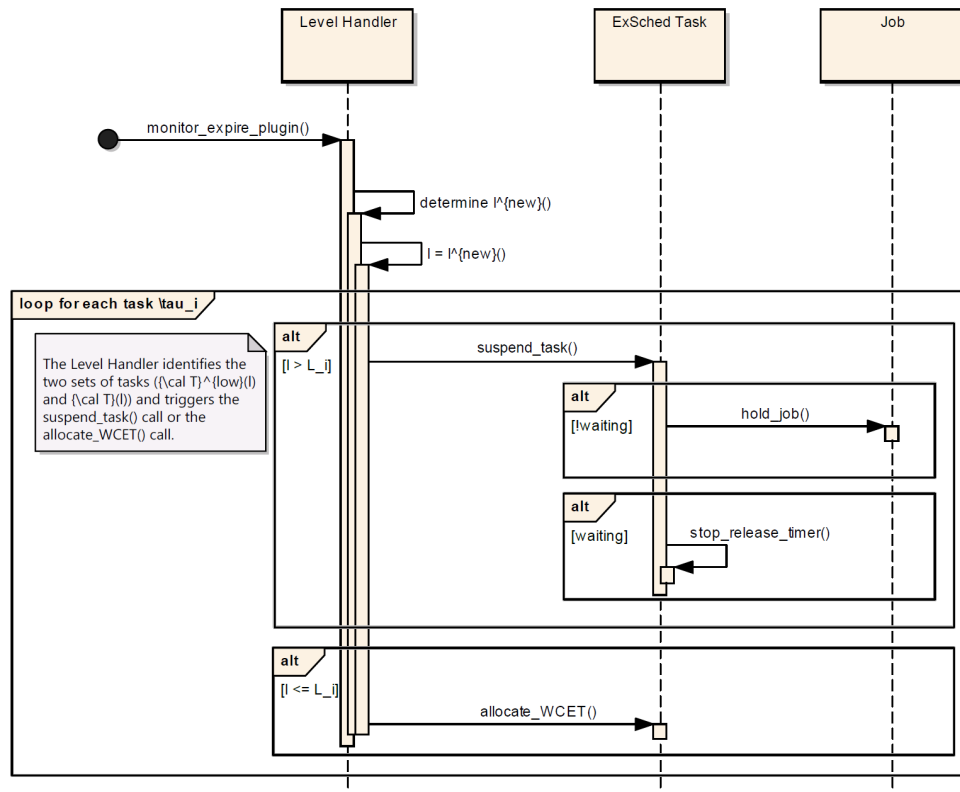


Figure 7.5: Sequence diagram of a criticality **level-up** change representing the calls which trigger state changes in the three state diagrams.

In Figure 7.5 the triggering event is a job not completed within its $C_i(l)$ WCET. Thus, the monitor expires, leading to a call to the `monitor_expire_plugin()` method. This call is handled by the Level Handler and the new criticality level of the system l^{new} is determined. The criticality level of the system is changed to l^{new} by the Level Handler. All tasks are handled according to their representative criticality levels. If the representative criticality-level of the task is less than the system's new criticality level, the task is suspended otherwise the task is allocated its WCET according to the system's current criticality level. When an ExSched Task must be suspended, the ExSched Task takes actions based on the state of the Task. If the ExSched Task is in the Ready state, its job is not allowed to continue execution. The `hold_job()` event is triggered which is handled by the Job. If the ExSched Task is in the Waiting state, its release timer is stopped. As a consequence it will not release new jobs.

Level-down change : A criticality level-down change is presented in Figure 7.6.

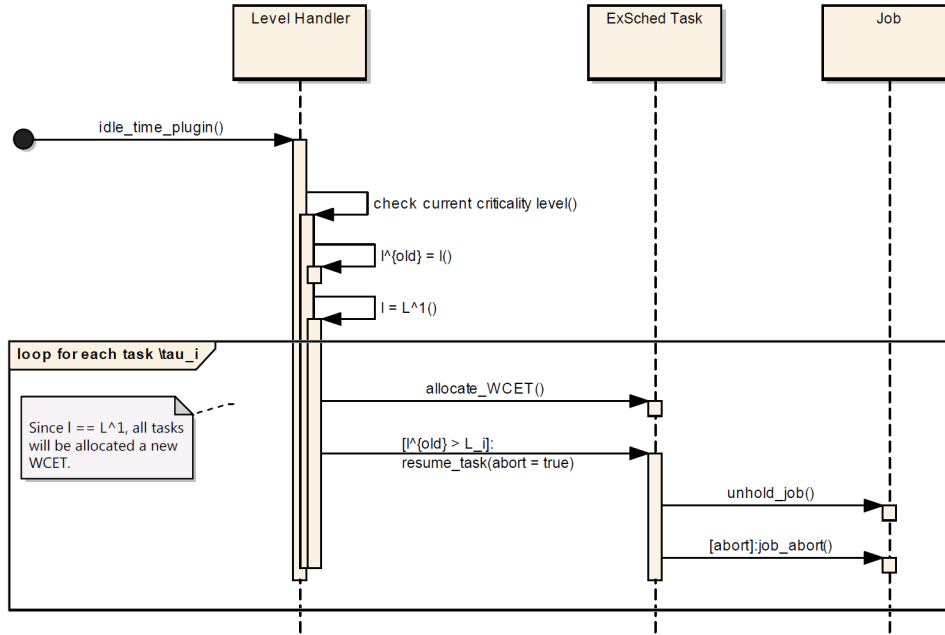


Figure 7.6: Sequence diagram of a criticality **level-down** change representing the calls which trigger state changes in the three state diagrams.

In Figure 7.6 the triggering event is the identification of an idle time. Thus the `idle_time_plugin()` method is called. This call is handled by the Level Handler. The Level Handler saves the old criticality level and inspects the current criticality level of the system. The Level Handler also changes the criticality level of the system to the lowest criticality level L^1 . Each task is allocated its respective $C_i(L^1)$ WCET. If the representative criticality level of the task is less than the old criticality level of the system l^{old} , meaning the task was suspended due to a level-up trigger, the task is allowed to resume with the `abort` parameter set to true. The job of the task is allowed to Contend for CPU time again using a call to the `unhold_job()` method. Since the `abort` parameter was true the job is aborted.

7.5 Additional design decisions

Upon occurrence of a monitor expire event we need to identify all the tasks that should be suspended. If a task has released a job and the task belongs to the identified set of tasks to be suspended (i.e., $\mathcal{T}^{low}(l^{new})$), irrespective of the state of the job, the job should be aborted. As mentioned earlier there is no notion of a job in the Linux Kernel. Hence, it is not possible to abort a particular job of a task. To abort a job of a task we decided to use Linux signals, in particular the POSIX compliant SIGUSR1 signal. Before a task is allowed to resume, if the task has a holding job, the task is sent the SIGUSR1 signal. The signal is handled by the task only when it is allowed to resume, i.e. upon an idle time. In this way, the user-space task is allowed to perform clean-up activities if required. The signal is not sent when the task was waiting to release another job, since the previous job is complete and there is no pending job to be aborted. However, its timer needs to be stopped so this is an additional action needed on an overrun.

Upon a level-down trigger, i.e. occurrence of an idle time, the release timer of all the suspended lower-criticality tasks must be restarted again. The next release time is calculated using Equation (5.1). If the lower-criticality task had a holding job, the job is aborted (explained above) and the task notifies the ExSched

Module that it is waiting for next release by calling the `rt_wait_for_period()` method from the ExSched Library. The `rt_wait_for_period()` is not called until a pending job is either finished or aborted.

7.5.1 Alternative: Killing and Restarting a Task

As an alternative to the job abort mechanism, a task could be killed and restarted. However, there are several reasons why this alternative is not a good choice as the following discussion shows.

In Linux it is possible to kill a task and then restart the task at a later moment. If a task is killed in Linux its state becomes ZOMBIE. The task loses its process ID after it is killed. Since a killed process cannot be revived again, the task must be restarted upon a criticality level-down trigger. During criticality-level changes, the ExSched Module can indicate which processes need be killed and restarted. A solution could be to make the user space ExSched Library take care of this. This means that the ExSched Library has to be aware of the user space task and must map it with its process ID. The ExSched Library must also be capable of restarting it. Although restarting a task is easy (the path of the executable is sufficient to restart a task), a disadvantage is the overhead of restarting a task. When a task is restarted it will re-initialize all its timing properties. The overhead of killing and restarting a task is significant. Restarting a task happens in a thread context and the cost of restarting the task is significantly higher than simply aborting a particular job. The clean-up activities required after killing a task have considerable costs as well.

A second disadvantage of this approach is that it requires a tight coupling between the tasks and the ExSched Library. As a consequence the ExSched Library would not just be a Library any more. The purpose of the ExSched Library is to facilitate the calls to the kernel space and still be independent of the tasks accessing the Library. For this scheme, the calls are not unidirectional any more but become bidirectional. An alternative could be to develop another user space application to handle killing and restarting tasks.

7.6 Process view

7.6.1 Timers and Interrupts

In this project we deal with timer interrupts only; all other interrupts are handled by the Linux Kernel. We use two types of timers in the ExSched Framework. The first type is a timer which works with jiffies. The second is a high resolution timer which works with nanosecond precision. The implementation of a periodic job release by the ExSched framework uses the jiffy timer. Each task is associated with an individual timer to release a new job. For monitoring the execution time of each job, we use high resolution timers (as stated above). Both job release and monitor expiration are timer interrupts. The job release interrupt is a result of release timer expiration and is handled by the `job_release()` method of the ExSched Module. The monitor expiration is a result of a job exceeding its allowed WCET and is handled by the `monitor_expire_handler()` method of the ExSched Module.

In Linux, interrupt handling is done in two halves. The so-called top half is the routine that actually responds to the interrupt, i.e., the one registered with `request_irq`. The bottom half is a routine that is scheduled by the top half to be executed later, outside the context of the interrupt handler. The timer interrupts are executed in the bottom half. This is because they may take more time to execute than the interrupts handled in the top half. Criticality level changes are also executed in the bottom half.

7.6.2 Race Conditions

Since handling criticality level changes is the key concern here, we must deal with race conditions arising due to this. While a criticality level change is being performed, a task may release a new job, stated differently a task may change its state from Waiting to Ready. This change in state may cause issues since the actions required to prevent the task and its job from executing further differ depending on the state it is in. To prevent these issues, interrupts are disabled during the criticality-level changes.

Level-Up change : The main concern during and after a criticality level-up trigger is that CPU time should not be allocated to a lower criticality (i.e. $L_i < l^{new}$) job. When the monitor expires, i.e., a job is not

finished within its $C_i(l)$ WCET, a level-up trigger is raised. While a level-up trigger is being handled, tasks may release new jobs. This may lead to unwanted effects, since the state of a task may change from Waiting to Ready while the AMC* plug-in is performing the criticality level change. The released job must be prevented from contending for CPU time, as this causes an overload situation. It was decided to prevent tasks from releasing new jobs by disabling all interrupts during a criticality level-up change. The interrupts are saved and restored later. While interrupts are disabled, in a single core processor, the jiffies are not updated. Therefore the release timers will not expire until the interrupts are restored. Note that low-criticality Waiting tasks are suspended during a level-up change. The process of suspending a task stops its release timer as well. Hence, when the interrupts are restored only high-criticality tasks will release new jobs, since the timers of the low-criticality tasks have already been stopped.

Level-Down change : The main concern during and after a criticality level-down trigger is that all pending jobs must be aborted before tasks are allowed to release new jobs. When an idle time is identified we use a similar approach as above. While a level-down change is being handled, we disable and store all interrupts and later restore and re-enable all interrupts. Therefore, all waiting tasks will release new jobs when the jiffies are updated after the interrupts are re-enabled.

Other than the race conditions discussed above there is one more race condition of releasing specific number of jobs by a task. This is discussed in the implementation chapter.

7.7 Scenarios

The scenarios discussed in the previous chapter remain the same although due to our design influencers (see Section 7.1) a difference is that job abortion is handled upon a level-down trigger rather than upon a level-up trigger. We illustrate the change with help of the scenario presented earlier in Section 5.4.2. The task set remains the same (Table 5.3). The updated scenario is presented in Figure 7.7.

First level-up trigger: Case 2 :

- At time 57ms, the current criticality level of the system is $l = L^1$
- Task 2 does not complete within its L^1 WCET.
- The new criticality level of the system is $l^{new} = L^2$
- The currently running job of Task 2 is allowed to continue execution for 4ms.
- Task 1 is suspended.

Second level-up trigger: Case 1 :

- At time 61ms the current criticality level of the system is $l = L^2$
- Task 2 does not complete within its L^2 WCET.
- The new criticality level of the system is $l^{new} = L^3$
- The currently running job of Task 2 is not allowed to continue execution. Task 2 is suspended.
- The contending job of Task 3 is not allowed to continue execution either. Task 3 is suspended.

Level-down trigger :

- At time 73ms the system goes back to criticality level L^1 .
- Task 1, Task 2 and Task 3 exit the suspended state.
- Task 1 takes the transition to the Waiting state and can release new jobs.
- Task 2 and Task 3 take the transition to the Ready state.
- The pending job of Task 2 is aborted.

- The pending job of Task 3 is aborted as well.
- Task 2 and Task 3 now take a transition to the waiting state.

The difference from the scenario presented in Section 5.4.2 is that the pending jobs of Task 2 and Task 3 are aborted at time 73ms (level-down change) instead of at time 61ms (level-up change).

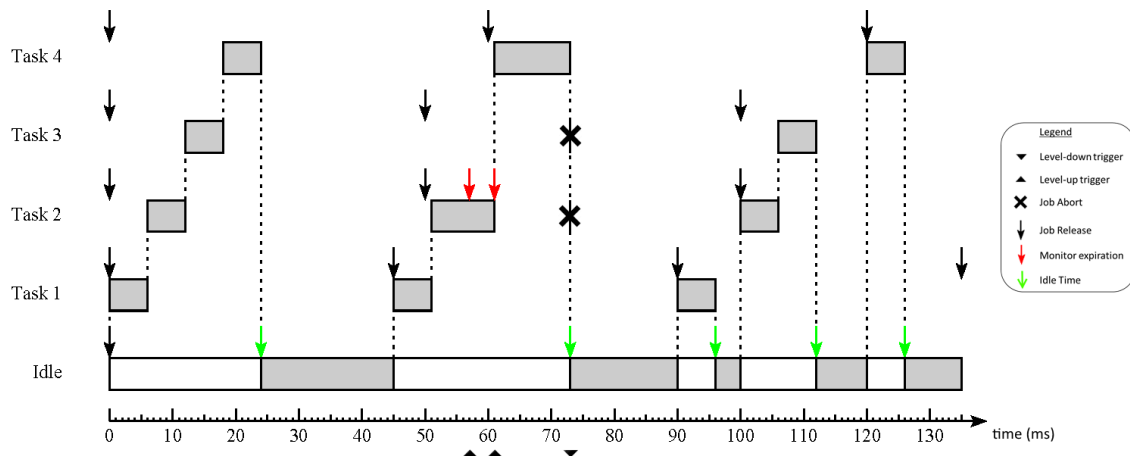


Figure 7.7: AMC* Scheme : Criticality Level-up change to L^2 and then to L^3 due to a task of representative criticality level L^2 and a level-down change due to a level- L^3 idle time

As mentioned before, a job has to abort itself. Therefore, job abortion has its own execution time. This execution time is not explicit from the figure above as the millisecond scale is not able to capture the execution time that is less than a tenth of a millisecond. Figure 7.8 depicts the behaviour at time 73ms. The figure shows that at time 73ms the respective jobs of Task 3 and Task 2 are allowed to execute but only to abort themselves. The jobs execute for a very short period of time as shown in the figure. The amount of time that the job executes depends on the speed of the processor.

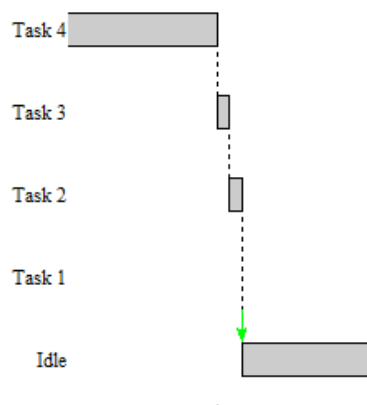


Figure 7.8: AMC* Scheme : Execution time of the job abort mechanism. In our set-up the execution time of job abort of Task 3 is less than one microsecond.

Chapter 8

Implementation

In this chapter, we discuss the implementation of the AMC* scheme in the ExSched framework. A plug-in has been developed to implement the AMC* scheme. The AMC* Plugin module is mainly responsible for:

- Storing information about WCETs of tasks per criticality level
- Storing the current criticality level of the system
- Handling the criticality level-up and level-down triggers

In addition, we have extended the framework (mainly the ExSched Module) with the support for the following generic functionality:

- Runtime Monitoring
- Detecting Idle Time
- Task Suspend-Resume
- Job Abort

The skeleton of ExSched task that resides in user space is extended as well to realize the job-abort functionality.

In Section 8.1, we discuss the Linux Scheduler and the API provided by the Linux kernel. In Section 8.2, we discuss the implementation of the Generic Functionality with which the ExSched Module is extended. In Section 8.2.4, we discuss the extension of the ExSched Task. Finally, in Section 8.3 the implementation of the AMC* Plugin Module is discussed.

8.1 Linux Scheduler and Kernel API

Here we discuss the Linux Kernel Scheduler and the API exposed by the Kernel. The ExSched framework encapsulates the Linux Scheduler.

A Linux task can be in one of the following five states. We refer to some of these states in the following section of this chapter.

1. `TASK_RUNNING` means that the task is in the Linux "Ready List".
2. `TASK_INTERRUPTIBLE` means that a task is waiting for a signal or a resource (sleeping).
3. `TASK_UNINTERRUPTIBLE` means that a task is waiting for a resource (sleeping); it is in the Linux "Wait Queue".
4. `TASK_ZOMBIE` means that the task is killed/dead and has not been purged yet. In Linux when a task dies, its task-descriptor stays in memory. While the task-descriptor is still in the memory the state of the task is `TASK_ZOMBIE`.
5. `TASK_STOPPED` means that a task is being debugged.

The scheduler of the ExSched framework encapsulates the `SCHED_FIFO` scheduler of the Linux Kernel. `SCHED_FIFO` breaks ties for tasks with the same priority level in a first-in-first-out fashion. For our implementation it is mandatory for each task to have a unique priority as dictated by the AMC* model, hence `SCHED_FIFO` does not play a role other than providing priority ordered scheduling.

The Linux kernel provides the `schedule()` method which is called by the ExSched module when a task indicates completion of a job by calling the `rt_wait_for_period()` method of the ExSched Library. The `schedule()` method puts the task to sleep until the Linux scheduler reaches the priority of the task which called the `schedule()` method. If the calling task has not released another job (its release timer has not expired yet), it will continue to sleep and the scheduler will continue scheduling other tasks. The `sched_setscheduler()` method sets the `SCHED_FIFO` scheduler to be used by the Linux Kernel for scheduling the ExSched tasks. The `set_tsk_need_resched()` (see below) method corresponds to the `hold_job()` method, which we presented as a symbolic representation in the design chapter. Similarly the `clear_tsk_need_resched()` (see below) method corresponds to the `unhold_job()` method. The `wake_up_process()` method sets the task state to `TASK_RUNNING` and puts the task back in the ready queue of the Linux Kernel. (Table 8.1) summarizes the functionality of these methods.

Table 8.1: Kernel-API for ExSched Module.

Method	Description
<code>schedule()</code>	A call to this method puts the task to sleep until the Linux scheduler reaches the priority of the task which called the <code>schedule()</code> method. If the calling task has not released another job, it will continue to sleep and the scheduler will continue scheduling other tasks.
<code>sched_setscheduler()</code>	This method is used to indicate the Linux Kernel to assign the scheduler type to the task thread. (As mentioned earlier, in ExSched the scheduler type is always <code>SCHED_FIFO</code> .)
<code>set_tsk_need_resched(task)</code>	This method sets the <code>NEED_RESCHED</code> flag in the Linux task structure. The Linux scheduler then waits for the next instance of the <code>schedule()</code> call. The parameter is a pointer to the Linux task structure.
<code>clear_tsk_need_resched(task)</code>	This method clears the <code>NEED_RESCHED</code> flag in the Linux task structure. The parameter is a pointer to the Linux task structure.
<code>wake_up_process(task)</code>	This method sets the task state to <code>TASK_RUNNING</code> and puts the task back in the ready queue of the Linux Kernel. The parameter is a pointer to the Linux task structure.

8.2 Generic Functionality

In this section, we discuss the extension of the ExSched module that implements the generic functionality for runtime monitoring and detecting idle time. We also discuss the implementation of task suspend-resume and job abort. In Section 8.2.1, we discuss the functionality of runtime monitoring. In Section 8.2.2, we discuss the functionality to detect idle time. We discuss the functionality of task suspend-resume and job abort in Sections 8.2.3 and 8.2.4 respectively.

8.2.1 Runtime Monitoring

To efficiently implement runtime monitoring, the key is to identify when the CPU is allocated to a job and when it is deallocated. As discussed in the previous chapter, in Table 6.1 we identified two methods which are responsible for allocating and deallocating the CPU i.e. `preempt_curr()` and `preempt_switch()`. Just before the CPU is allocated to the a job we call the `start_monitor_timer()` method. As soon as the CPU is deallocated, the `stop_monitor_timer()` method is called. The `start_monitor_timer()` and `stop_monitor_timer()` methods have been summarized in the design chapter in Table 6.5. The method `preempt_curr()` is responsible for deallocating the CPU from the current job and allocating it to a higher priority job. Therefore, the `preempt_curr()` method calls the `stop_monitor_timer()` method for the current job on the CPU and the `start_monitor_timer()` method for the higher priority job to which the CPU is allocated next. Similarly, the method `preempt_switch()` is responsible for calling the `stop_monitor_timer()` method for the current job on the CPU. If the ready queue is not empty, the `start_monitor_timer()` method will be called for the highest priority job in ExSched's ready queue. The CPU will be allocated to this job for execution.

8.2.2 Detecting Idle Time

As mentioned earlier, the job of a task is stored in a priority-ordered ready queue. If the ready queue is empty, no jobs are contending for CPU time. Therefore, an idle time is detected. We check the number of tasks in the ready queue at two points. One is when a job signals completion and the other is after a criticality

level-up change is performed (see Scenario 6 in Section 5.4.6). Once an idle time is detected, the registered callback method of the plug-in is invoked, i.e. the `idle_time_plugin()` method.

We present the functionality of Runtime Monitoring and detecting Idle Time with the help of a sequence diagram in Figure 8.1. When a job completes, its monitor timer is stopped and the task is dequeued from the ExSched's ready queue and the ready queue is inspected. If the ready queue is empty, an idle time is detected which invokes the AMC* Plug-in. After the plug-in has handled the `idle_time_plugin()` call, the task waits for its next periodic release by calling the `mod_timer()` and `schedule()` methods. When the release timer of the task expires, the task is added to the ExSched's ready queue for execution. After all other higher priority tasks are executed, the `schedule()` method returns control back to the task. The monitor timer is started and the job starts to run. If the monitor timer expires, the AMC* Plug-in is invoked by a call to the `monitor_expire_plugin()` method. After the plug-in has handled the `monitor_expire_plugin()` call, the ready queue is inspected again. If the ready queue is empty, an idle time is detected which invokes the AMC* Plug-in. Notice that the methods `local_irq_save()` and `local_irq_restore()` are called before and after the `monitor_expire_plugin()` and `idle_time_plugin()` methods. This is to ensure that all interrupts are disabled before a criticality-level change and restored later.

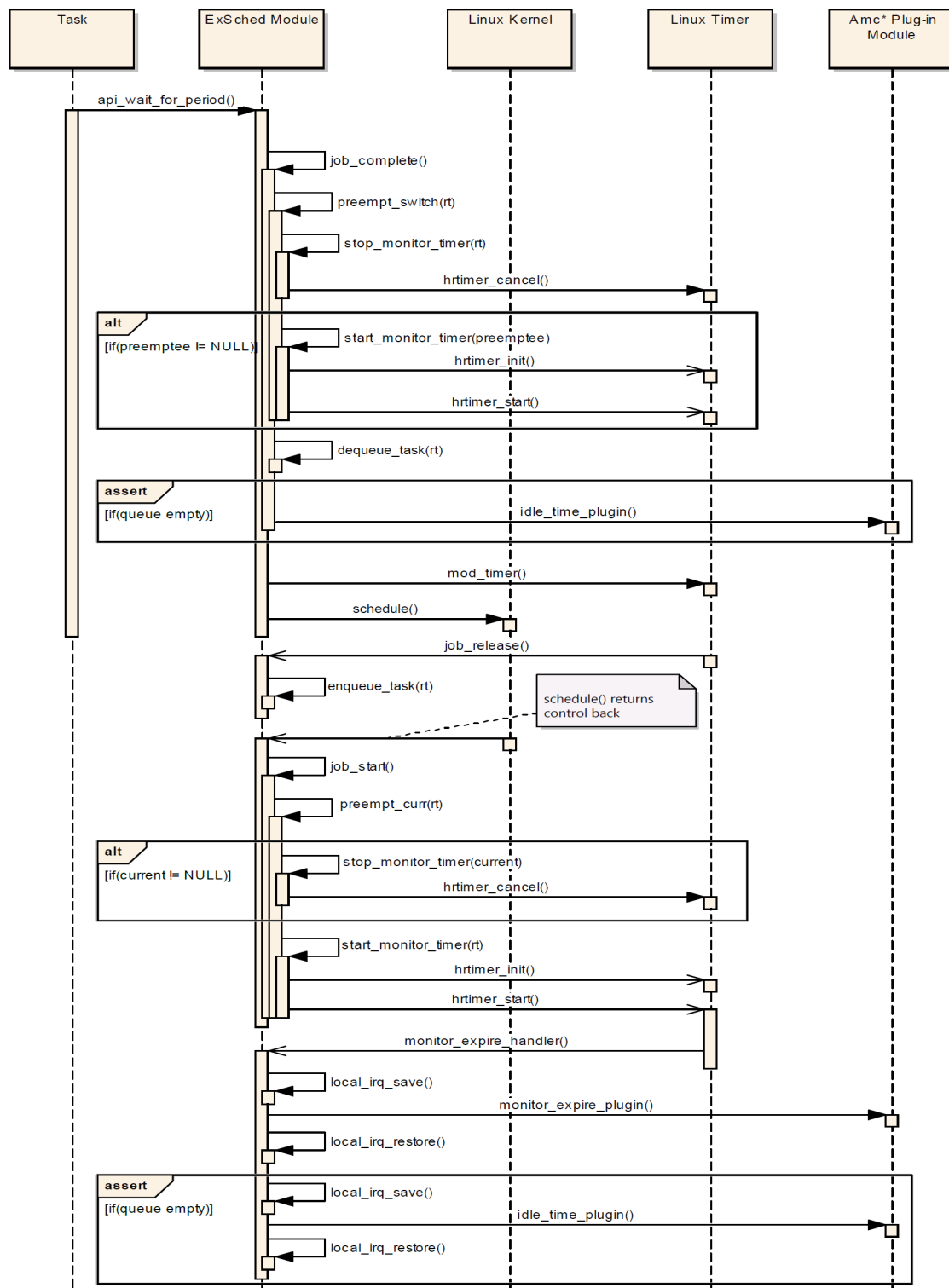


Figure 8.1: Sequence diagram representing the functionality of Runtime Monitoring and detecting Idle Time.

8.2.3 Task Suspend-Resume

Although we have extended the ExSched module to provide the suspend-resume functionality, the implementation is inspired by the HSF already developed as a plug-in for the ExSched framework (see Section 6.3). In the HSF when the budget of a server expires, all its tasks are suspended and its tasks are resumed when the budget of the server is replenished. We used this behaviour as a starting point for our extension. To implement the suspend-resume functionality at the task level, we use the task states defined by the Linux task structure namely the `TASK_RUNNING` and `TASK_UNINTERRUPTIBLE` states. When the task is running, the state of the task is `TASK_RUNNING` and when a task is suspended, the state is `TASK_UNINTERRUPTIBLE`. There is a third state in the Linux task structure, namely `TASK_INTERRUPTIBLE`. We do not use this state since a task in this state can be woken up by any signal which may lead to issues in scheduling the task.

We must ensure that previously suspended tasks are not suspended again. When the AMC* plug-in module executes a criticality level-up change, all tasks in \mathcal{T}^{low} (Equation (3.2)) must be suspended. From the set \mathcal{T}^{low} , tasks may be in either of the three states Waiting, Ready and Suspended. The tasks which are already suspended (upon a previous criticality level change) will not be suspended again or even revisited for suspension; this is discussed in Section 8.3.1.

Apart from the methods discussed in Table 6.1, there are two methods local to the ExSched module to allocate and deallocate CPU to a job namely `preempt_out()` and `preempt_in()`. These methods are described in Table 8.2 and are referred to in the next sections of the report.

Table 8.2: Methods of the ExSched Module which allocate or deallocate the CPU to a job.

Method	Description
<code>preempt_out(resch_task_t*)</code>	The methods deallocates CPU from a job of a task; a pointer to the task is supplied as a parameter.
<code>preempt_in(resch_task_rt*)</code>	The method allocates CPU to a job of a task; a pointer to the task is supplied as a parameter.

Suspend Task

There are two states of an ExSched task that need to be considered for implementing the suspend functionality, namely Waiting and Ready.

Waiting : When a task is in the waiting state its release timer is running. When a task is suspended with a call of `suspend_task()`, its release timer must be stopped else the task would release a new job upon timer expiration. The release timer is stopped by a call to the `stop_release_timer()` method. Note that when task is in the Waiting state, the state in the Linux task structure is `TASK_UNINTERRUPTIBLE`, hence we do not need to change the state in the Linux structure.

Ready : When a task is in the Ready state, the state in the Linux task structure is `TASK_RUNNING`. To suspend a task in the `TASK_RUNNING` state, a number of actions are needed. First, the task's state in Linux needs to be changed to `TASK_UNINTERRUPTIBLE` so that it is not scheduled by the Linux scheduler. Then, the Linux scheduler needs to be informed that the task needs to be held from further execution. The CPU must be deallocated from the task and finally the task must be dequeued from the ready queue. Thus when a task is in the Running state and the `suspend_task()` method is called, the state in the Linux task structure is set to `TASK_UNINTERRUPTIBLE`. The `NEED_RESCHED` flag is set by calling the method `set_tsk_need_resched()` method which prevents the job from executing further. The CPU is deallocated from the task with the help of the `preempt_out()` method call. Finally, the task is dequeued from the ready queue of the ExSched module by calling the method `dequeue_task()`. When these methods are called, the Linux Kernel will not consider this task for scheduling and the ExSched task is suspended. While a task is in the running state, the release timer associated with the task is not running, therefore after suspension no new job will be released by the task.

Pseudo code for the `suspend_task()` method is:

```
void suspend_task(resch_task_t* rt){
    if(rt->waiting == true){
        stop_release_timer(rt)
    } else {
        rt->task->state = TASK_UNINTERRUPTIBLE;
        set_tsk_need_resched(rt->task);
        preempt_out(rt);
        dequeue_task(rt);
    }
}
```

Resume Task

To implement the resume functionality, it is necessary to know the state of the task before it was suspended. To handle this we introduced the waiting flag in the ExSched task structure. Once the `resume_task()` method is called, the task moves to either the Waiting or the Ready state.

Waiting : When the `resume_task()` method is called and if the waiting flag in the ExSched task structure is true the task moves from the Suspended state to the Waiting state. A new release time is identified and the release timer of the task is started according to the next release time. The release time of the previous job and the period of the task are stored in the ExSched task structure; the next release time is calculated from Equation (5.1). Note that the state of the task in the Linux task structure is not changed since the next expiration of the periodic timer triggers the `job_release()` method which changes the state of the task in the Linux task structure from `TASK_UNINTERRUPTIBLE` to `TASK_RUNNING`.

Ready : If the waiting flag in the ExSched task structure is false and the `resume_task()` method is called, the task moves from the Suspended state to the Ready state. To change the state back to Ready, a number of actions are needed. First, the task needs to be sent the abort signal before it is resumed. Then, the Linux scheduler must be notified that the task is allowed to continue execution by a call to `clear_tsk_need_resched()`. The task must be enqueued back in the ready queue with the `enqueue_task()` method call. The state of the Linux task needs to be changed from `TASK_INTERRUPTIBLE` to `TASK_RUNNING` by calling the `wake_up_process()` method. Finally, the CPU must be allocated to the task for execution. The task is signalled to abort by the `send_sig()` call. The job is allowed to resume when the CPU is allocated to the job by a call to the `preempt_in()` method.

By inspecting the waiting flag before the task is resumed we ensure that the abort signal is sent only to tasks that were in the Ready state before the Suspended state. Pseudo code for the `resume_task()` method is:

```
void resume_task(resch_task_t* rt, bool abort){
    if(rt->waiting == true){
        start_release_timer(rt);
    } else {
        if(abort==true){
            send_sig(rt->pid, SIGUSR1);
        }
        clear_tsk_need_resched(rt->task);
        enqueue_task(rt);
        wake_up_process(rt->task);
        preempt_in(rt);
    }
}
```

8.2.4 Job Abort

The ExSched framework and the Linux kernel do not provide primitives which would allow abortion of a particular job of an ExSched task. As mentioned in the Design chapter, we use Linux signals for implementing the Job Abort functionality. To abort the job of a task the SIGUSR1 signal is used.

As discussed earlier in Section 7.1, the key design influencer is to reduce the overhead during the criticality level-up change. To this end, the SIGUSR1 signal is delayed to a criticality level-down change, i.e., an idle time, before the task is allowed to resume and the job is aborted thereafter. Our decision to send a signal conforms to the design influencer, since the signal is handled and the job is aborted only when the task is allowed to resume upon occurrence of an idle time.

A constraint on such a design, i.e. using signals, is that aborting a job must be handled by the tasks in user space. If the signals are not handled properly, this may lead to undesired system behaviour like another criticality level-up trigger occurring immediately. Care must be taken by system engineers and developers that the signal is not masked and is handled properly, i.e. the pending job is indeed aborted.

Since abortion happens upon a criticality level-down change, all tasks in the Suspended state must be resumed, and all pending jobs must be aborted. The waiting flag in the ExSched task control block helps to achieve this. Upon a criticality level-down change, the waiting flag is inspected for each of the tasks in the Suspended state. If the waiting flag is set to true, the task was in the Waiting state before being suspended. These tasks will not be sent the abort signal since there are no pending jobs to be aborted. Instead, the time of the next job release is calculated and the release timer of the task is started. If the waiting flag was set to false, the task was in the Running state before being suspended. These tasks are sent the SIGUSR1 abort signal, since they have unfinished jobs which need to be aborted. Once the abort signal is handled by the task and the job is aborted, the `rt_wait_for_period()` method is called. Thus, the task will release next job according to its period. Note that depending on the duration for which a task is suspended the task may have missed several job releases. This is expected behaviour as defined by the AMC* scheme.

The structure of ExSched task needs to be extended in order to handle the Job Abort signal. This structure and the extension of this structure needs to be discussed in more detail to further understand the implementation of job abortion.

Original ExSched Task

In ExSched, a task runs on the main thread; each job of the task executes inside a for loop. The increment of the loop counter represents a new job of the task. A task is allocated an ExSched ID when submitted for real-time scheduling and a Linux Process ID when a task starts execution. Both IDs are stored in the task control block of the ExSched Task. The sample code of an ExSched task is shown in Figure 8.2.

```
main(timeval C, timeval T, timeval D, int prio, int nr_jobs ){
    int i;
    rt_init();
    rt_set_wcet(C);
    rt_set_deadline(D);
    rt_set_period(T);
    rt_set_priority(prio);
    rt_run(0);
    for(i=0;i<nr_jobs;i++)
    {
        do_work();
        rt_wait_for_period();
    }
    rt_exit();
}
```

Figure 8.2: User space ExSched Task

The task is initialized in the ExSched module by a call to the `rt_init()` method. The task supplies its timing parameters using the `rt_set_wcet(C)`, `rt_set_deadline(D)`, `rt_set_period(T)` and `rt_set_priority(prio)` methods. The task is submitted for scheduling by calling the `rt_run()` method. A for loop is responsible for iterating jobs of the task. The `do_work()` method represents a job. A task will not release a new job unless its previous job is complete, because `rt_wait_for_period()` will not be called until `do_work()` has finished execution. In the ExSched framework, for FPPS, the deadline of a task is equal to the period of the task. Hence, the `rt_set_deadline()` API call is not relevant in our design and implementation.

Extension of the ExSched Task

To realize the AMC* scheme, the task skeleton of the ExSched Task must be modified (Figure 8.3). This modification realizes the job abort functionality. To abort a job, we use the `setjmp()` and `longjmp()` methods defined in the `setjmp.h` header file of the C standard library which provide "non-local jumps". These methods save and restore the processor states. The `setjmp(jmp_buf env)` method sets up the local `jmp_buf` buffer and initializes it for the jump. This method saves the program's calling environment in the environment buffer specified by the `env` argument for later use by the `longjmp()` method. If the return is from a direct invocation, `setjmp()` returns 0. If the return is from a call to `longjmp`, `setjmp` returns a non-zero value. The `longjmp(jmp_buf env, int value)` restores the context of the environment buffer `env` that was saved by the last invocation of the `setjmp()` method. The value specified by `value` is passed from `longjmp()` to `setjmp()`. After `longjmp()` is completed, program execution continues as if the corresponding invocation of `setjmp()` had just returned from its last actual call.

A global variable `JumpBuffer` is declared for each task. For each job, `setjmp()` is called to indicate the initialization point. The `SIGUSR1` signal is sent by the ExSched Module before the task is woken up by `wake_up_process()` method call. As soon as the job resumes execution the `SIGUSR1` signal is received. Upon receiving the signal, the normal flow of execution in the `do_work()` method is interrupted and `job_abort_handler()` is invoked. From the `job_abort_handler()` method, the `longjmp()` method is called and the control flow switches to the initialization point at line 26 in Figure 8.3. The `if` condition will fail since the return value of `setjmp()` is 1 and the `rt_wait_for_period()` method is called. This behaviour will happen consistently irrespective of the execution flow within the `do_work()` method. Now we consider the following two boundary cases.

- When the task is submitted for real-time scheduling by a call to the `rt_run()` method, it may not immediately start execution with its first job due to its priority (a low-priority task is scheduled after the high-priority tasks). In the mean time it may receive the Abort Signal. The `for` loop may not have been executed even once. At this point the `JumpBuffer` is not initialized by the task using the `setjmp()` method. The behaviour in Linux is undefined for this case and may lead to unexpected behaviour. Hence, we initialize the `JumpBuffer` at line 21 as well. As a consequence, the task may be re-submitted for real-time scheduling by the `rt_run(0)` call at line 24, which will cause no harm and the task will wait for next period.
- When a job of a task is in the Contending for CPU state, i.e., the job has been released but has not been allocated the CPU yet, it may receive the Abort Signal. At this point the loop counter has not been incremented and `longjmp()` method is called which forces the program to continue execution from the initialization point at line 26. This causes the loop counter to be inconsistent depending on the number of times this case has occurred. For a finite `nr_jobs`, the program counter could be at the beginning or at the end of the `do_work()` method. So it is impossible to know how many times it has executed. In most real-time applications a task may release infinite number of jobs. So, the value of loop counter is usually unimportant.

The extended task skeleton is presented in Figure 8.3.

```

1  jmp_buf JumpBuffer;
2
3  static void job_abort_handler(int signum){
4      longjmp(JumpBuffer, 1);
5  }
6
7  int main(int argc, char* argv[])
8  {
9      struct sigaction abort_sig;
10     /* C, T, prio and nr_jobs are parsed from the arguments supplied
        while starting the task.*/
11     int i;
12     rt_init();
13     rt_set_wcet(C);
14     rt_set_period(T);
15     rt_set_priority(prio);
16     rt_set_rep_crit(rep_crit, unsigned_long[NO_OF_CRIT_LEVELS]);
17     memset(&abort_sig, 0, sizeof(abort_sig));
18     sigemptyset(&abort_sig.sa_mask);
19     abort_sig.sa_handler = job_abort_handler;
20     abort_sig.sa_flags = 0;
21     setjmp(JumpBuffer);
22     sigaction(SIGUSR1, &abort_sig, NULL);
23
24     rt_run(0);
25     for (i = 0; i < nr_jobs; i ++){
26         if(setjmp(JumpBuffer) == 0){
27             do_work();
28         }
29         rt_wait_for_period();
30     }
31     rt_exit();
32     return 0;
33 }

```

Figure 8.3: AMC* Task

Table 8.3: Variables and methods introduced in the Task Skeleton.

Variable or Method	Description
<code>int setjmp (jmp_buf env)</code>	Sets up the local <code>jmp_buf</code> buffer and initializes it for the jump. This method saves the program's calling environment in the environment buffer specified by the <code>env</code> argument for later use by <code>longjmp</code> . If the return is from a direct invocation, <code>setjmp</code> returns 0. If the return is from a call to <code>longjmp</code> , <code>setjmp</code> returns a non-zero value.
<code>void longjmp (jmp_buf env, int value)</code>	Restores the context of the environment buffer <code>env</code> that was saved by invocation of the <code>setjmp()</code> method in the same invocation of the program. The value specified by <code>value</code> is passed from <code>longjmp()</code> to <code>setjmp()</code> . After <code>longjmp()</code> is completed, program execution continues as if the corresponding invocation of <code>setjmp()</code> had just returned.
<code>JumpBuffer</code>	This is a <code>jmp_buf</code> type variable initialized by the <code>setjmp()</code> method in line 32.
<code>abort_sig</code>	This is the variable of type <code>sigaction</code> that is registered with the <code>SI-GUSR1</code> signal. The handler method associated with the <code>sigaction</code> is <code>job_abort_handler()</code> .
<code>job_abort_handler()</code>	This method is executed when a job is allocated the CPU for execution and the job needs to be aborted. This method is executed only if a job abort signal was sent by the <code>ExSched</code> module. The <code>job_abort_handler()</code> method is registered as a handler of the <code>SI-GUSR1</code> signal.

8.3 AMC* Plugin Module

In this section, we discuss the implementation of the plug-in module developed to realize the AMC* scheme. The plug-in module stores the information about the representative criticality level of the tasks and the WCET value per criticality level. The module is also responsible for maintaining and changing criticality levels of the task and for aborting tasks. In Section 8.3.1 we discuss how the criticality-associated information of a task is stored. Then in Section 8.3.2 we discuss preventing race conditions from occurring.

8.3.1 Storing Task Information

To store all the tasks we define an array `task_rep_crit[NO_OF_CRIT_LEVELS]` indexed according to the number of criticality levels supported. Each index stores a list of ExSched task pointers. The pointer to an ExSched task is fetched using the `get_current_task()` method (see Table 7.2). The lists only contain tasks with representative criticality level corresponding to the array index. Hence, within the array the tasks are sorted according to their representative criticality levels. We use a list to store the task pointers instead of a static array because tasks can be dynamically added and removed in the ExSched framework. Upon initialization it is not known how many tasks have the same representative criticality level. So a fixed-size array cannot be used. If we use a fixed size array it must be of size `MAX_RT_TASKS` which is the maximum number of tasks supported. This would require a substantial amount of memory given the number of criticality levels supported by the system and the maximum number of tasks supported by the system. Therefore, a fixed size array is not used for storing the task pointers in a sorted manner. The following shows the array defined within the AMC* Plugin Module to store the ExSched task pointers (for more information on Linked Lists in Linux Kernel see [18]):

```
struct amc_task {
    struct list_head task_list;
    resch_task_t* rt;
};

struct list_head task_rep_crit[NO_OF_CRIT_LEVELS];
```

The AMC* Plugin module maintains the current criticality level of the system in a variable `curr_crit_level`. Criticality-level changes are made based on the task that triggers a criticality-level change. A new criticality level is identified based on the current criticality level, the representative criticality level of the triggering task and the WCET of the triggering task.

For quick access to the WCETs of a triggering task we store the WCETs of the tasks in a static 2D array `task_wcet[MAX_RT_TASKS][NO_OF_CRIT_LEVELS]` indexed according to the ExSched ID of the task. Each index stores the WCET per criticality level of the task represented by the index. ExSched IDs are numeric starting with zero and sequentially assigned to each task submitted to ExSched with an increment of one. This makes storing and accessing the WCETs of a task to determine the new criticality level of the system easy and quick. It is important to make sure that at criticality levels higher than the representative criticality level the WCET is zero. This is handled in the `rt_set_rep_crit()` method. The following shows the array defined to store the WCETs of a task:

```
unsigned long task_wcet[MAX_RT_TASKS][NO_OF_CRIT_LEVELS];
```

When the monitor expires, the new criticality level of the system needs to be determined. The new criticality level is determined on the basis of the current criticality level of the system and the AMC*-related properties of the task like the representative criticality level and the WCET per criticality-level. For this, we first check that the current criticality level of the system is not the highest criticality level otherwise it is an Error Condition. Then we iterate through the WCETs stored at the index of the triggering task in the `task_wcet` array starting with the value at the index `curr_crit_level + 1`. If the WCET at this index is higher than the current WCET of the task, we have determined the new criticality level of the system. Otherwise, we iterate further until we find a WCET higher than the current WCET of the task. If no match is found, we check if the WCET of a task is zero at a higher criticality level. If this is the case, the first occurrence of a zero WCET determines the new criticality level otherwise we have encountered an Error Condition. Pseudo code for this functionality is:

```
void determine_new() {
    int i = 0;
    int new_crit_level = 0;
    for (i = curr_crit_level + 1; i <= NO_OF_CRIT_LEVELS; i++){
        if (task_wcet[rt->ID][i]==0){
            new_crit_level = i;
            break;
        } elseif (task_wcet[rt->ID][i] == rt->wcet){
            continue;
        } elseif (task_wcet[rt->ID][i] > rt->wcet){
            new_crit_level = i;
            break;
        }
    }
    if(new_crit_level == 0){
        //Error Condition
    } else {
        level_up(new_crit_level);
    }
}
```

Level-Up Change

For executing a criticality level-up change we use the `task_rep_crit` array. We present the functionality of handling a criticality level-up change with the help of a sequence diagram in Figure 8.4. The AMC* Plug-in module is invoked by the callback function `monitor_expire_plugin()`. The `monitor_expire_plugin()` method first invokes the `determine_new()` method followed by a call to the `level_up()` method. In this method we iterate through the array starting at the index of `curr_crit_level`. Up to the index of `new_crit_level`, all tasks in the list are suspended. Starting from the index `new_crit_level`, all tasks in the list are allocated a WCET according to the `task_wcet` array. Note here that lower criticality tasks which may have already been suspended are not revisited again. Pseudo code for this functionality is presented below:

```
void level_up(int new_crit_level){
    int i = 0;
    for (i = curr_crit_level; i <= NO_OF_CRIT_LEVELS; i++){
        if (i < new_crit_level){
            foreach(resch_task_t* t in task_rep_crit[i]){
                suspend_task(t);
            }
        } elseif(i >= new_crit_level) {
            foreach(resch_task_t* t in task_rep_crit[i]){
                //allocate_WCET();
                rt->wcet = task_wcet[rt->ID][new_crit_level];
            }
        }
    }
    curr_crit_level = new_crit_level;
}
```

Figure 8.4 also shows the functionality of the `suspend_task()` method. When the method is called, the ExSched module inspects the waiting flag in the TCB. Based on the value of this flag, the state of the task is determined. Accordingly, either the job is not allowed to continue execution or the release timer of the task is stopped (see Suspend Task in Section 8.2.3).

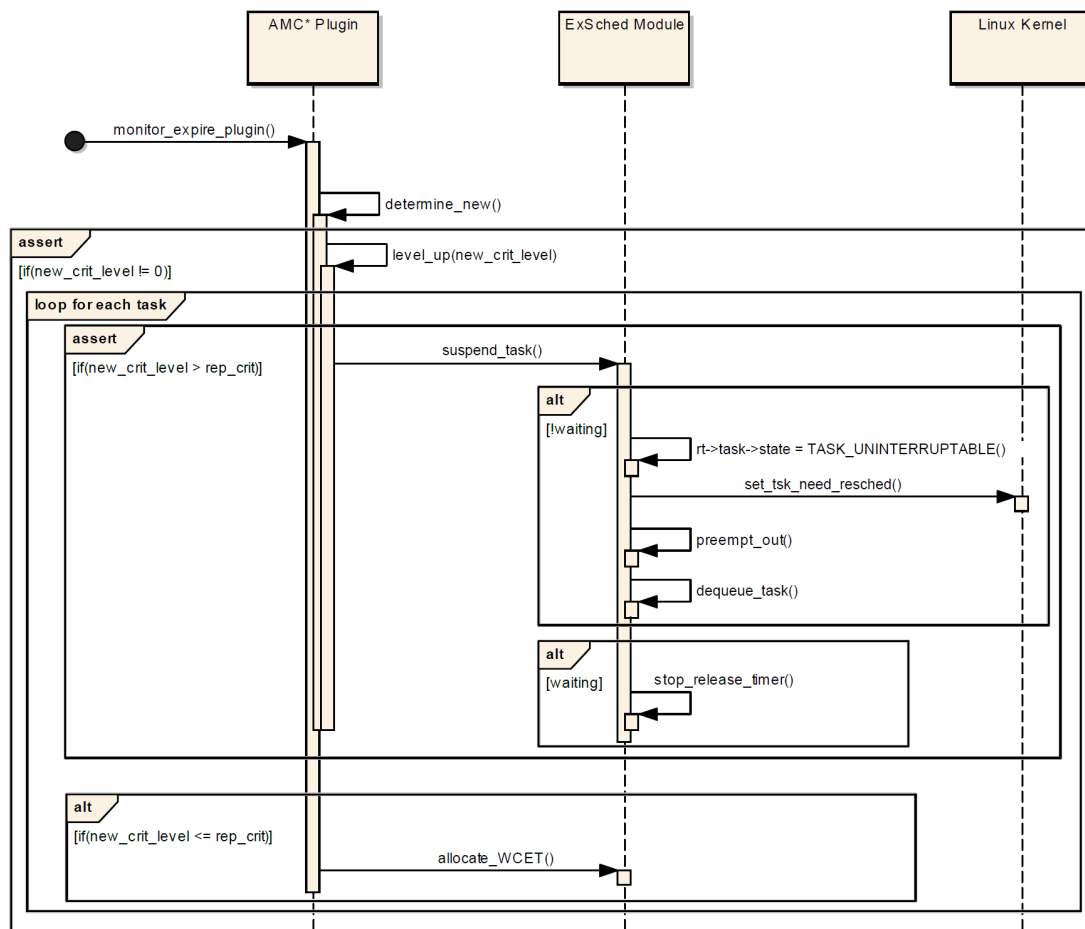


Figure 8.4: Sequence diagram for level-up change functionality.

Level-Down Change

To execute a criticality level-down change is simpler to explain than a level-up change. We present the functionality of handling a criticality level-down change with the help of a sequence diagram in Figure 8.5. In the figure the AMC* Plug-in module is invoked by the callback function `idle_time_plugin()`. In this method, the current criticality level is determined first, after which the `level_down()` method is called. In this method, all suspended tasks must be resumed and all tasks must be allocated WCETs corresponding to the lowest criticality level of the system. Pseudo code for this functionality is presented below:

```
void level_down(){
    int i = 0;
    for (i = 1; i <= NO_OF_CRIT_LEVELS; i++){
        if (i < curr_crit_level){
            foreach(resch_task_t* t in task_rep_crit[i]){
                resume_task(t,true);
                rt->wcet = task_wcet[rt->ID][1]; //allocate_WCET()
            }
        } elseif(i >= curr_crit_level) {
            foreach(resch_task_t* t in task_rep_crit[i]){
                rt->wcet = task_wcet[rt->ID][1]; //allocate_WCET()
            }
        }
    }
    curr_crit_level = 1;
}
```

Figure 8.5 also shows the functionality of the `resume_task()` method. When the method is called with the abort parameter set, the ExSched module inspects the waiting flag in the TCB. Based on the value of this flag, the state of the task before it was suspended is determined. Accordingly, either the job is allowed to continue execution for aborting itself or the release timer of the tasks is started (see Resume Task in Section 8.2.3).

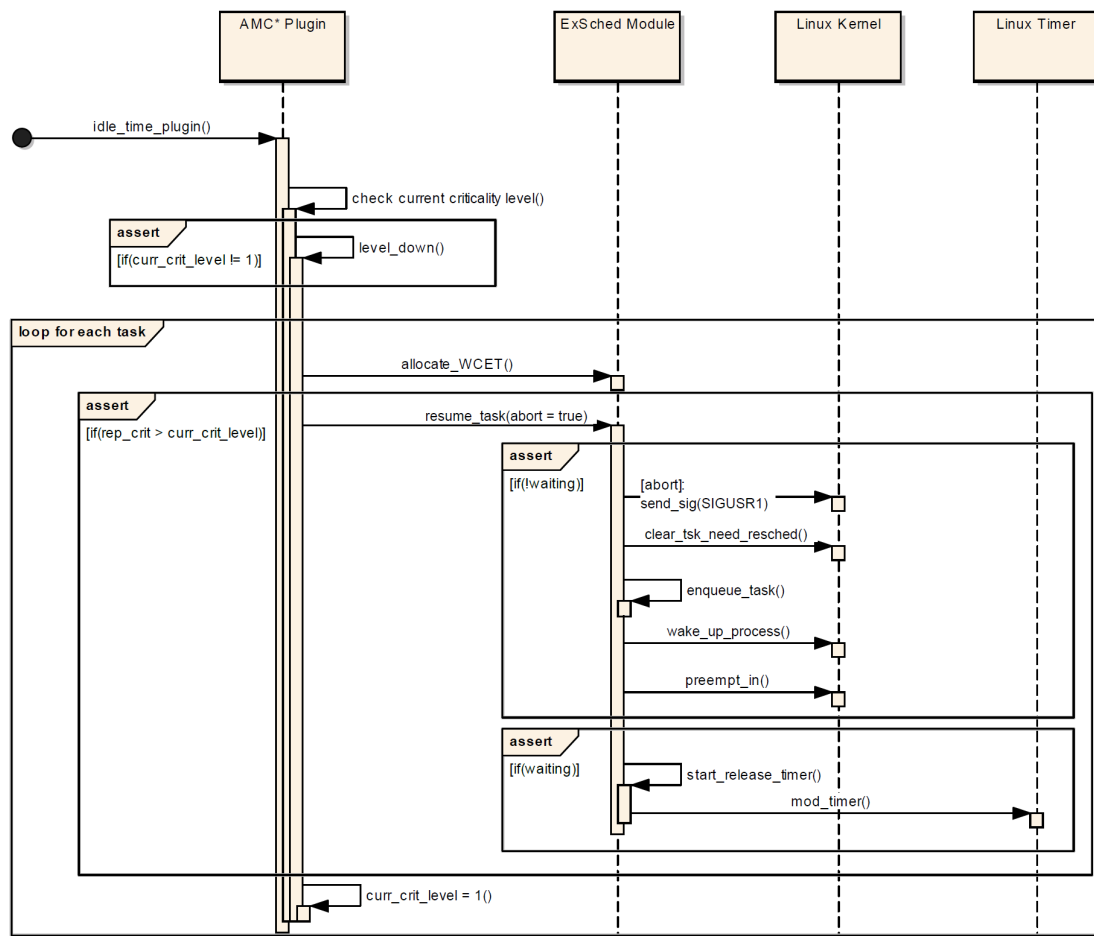


Figure 8.5: Sequence diagram for level-down change functionality.

8.3.2 Race Conditions

As discussed in the design chapter, to avoid occurrence of the race conditions we disable and store the interrupts before a criticality level change is performed and enable and restore them after the change is performed. The Linux Kernel provides two methods for achieving this, `local_irq_save()` and `local_irq_restore()`. When the `local_irq_save()` method is called, all interrupts are disabled and if an interrupt arrives it is stored. While the interrupts are disabled in a single core processor, the jiffies are not updated. Therefore, the tasks in the Waiting state will not release new jobs as the timers associated with the job will not expire. The release timers of tasks in the ready state are not active therefore these will not release new jobs either. Hence, there will be no state changes induced by a task while interrupts are disabled. We conclude that disabling interrupts is sufficient to avoid that race conditions arise. We enable the interrupts again after the criticality level change is performed.

Table 8.4: API to disable and enable the interrupts.

Method	Description
<code>local_irq_save(unsigned long flags)</code>	A call to this method disables interrupt delivery on the processor after saving the current interrupt state into flags.
<code>local_irq_restore(unsigned long flags)</code>	This method is used to restore the state which was stored in flags by <code>local_irq_save()</code> .

Chapter 9

Conclusions

This chapter reflects on the goals of i-Game and gives the conclusion of the achieved results. It gives recommendations for future work on the AMC* scheme and the ExSched framework in general.

9.1 Results and conclusions

This work was initiated to extend a real-time scheduler to support mechanisms for criticality-level changes. A future goal is to consider a hierarchical scheduling framework and multi-core scheduling. This report presents the first step towards achieving this goal. From different schemes for Mixed Criticality scheduling found in literature we considered several schemes and selected the AMC scheme for design and implementation. We have discussed a generalization of the AMC scheme which we called the AMC* Scheme in this report. We have discussed the semantics of the AMC* scheme and derived a set of scenarios which are sufficient to validate the scheme's behaviour.

For the design and implementation of the AMC* scheme, we considered three frameworks from which ExSched was chosen. ExSched supports hierarchical and multi-core scheduling. To limit the scope of the work and in view of the time, it was decided to design and implement the AMC* scheme for the FPPS scheduling keeping the aspects of hierarchical and multi-core scheduling for future work.

In this report, we presented a design for the AMC* scheme which conforms to the existing plug-in based design of the ExSched framework. We have extended the ExSched framework to support generic functionality for run-time monitoring, detecting idle time and task management services namely suspend-resume and job abort. We designed and implemented a plug-in for the AMC* scheme to handle criticality level changes. We have successfully run the set of scenarios against the implementation for validation of the design and implementation.

Due to the plug-in based design of the ExSched framework and the extended functionality, it would be possible to develop other mixed criticality schemes as well. However, for the schemes that require re-prioritization of tasks like the SMC scheme, the framework needs to be extended further. The ExSched framework provides a wide range of scheduling capabilities and is very suitable for prototyping purposes.

The deliverables of this work were verified against the functional and non-functional requirements as given in Chapter 4. In conclusion, the main goal of our work has been achieved and a real-time scheduler has been extended to provide support for the AMC* scheme.

9.2 Future work

The ExSched framework can be extended to provide support for changing task priorities at runtime. This will open the opportunity to develop, compare and analyse other mixed criticality schemes as well. The results of this work can be taken further to extend the HSF plug-in with the AMC* scheme. Before that can be done some questions need to be answered like how will criticality be associated with a *server*? Some of the options can be:

- Within a server, each task will have the same criticality level, i.e., a criticality level is assigned to a server. In this approach, a criticality level is changed only if a higher criticality server has unfinished tasks within its allocated budget. With a change in the system's criticality-level, the server of high-criticality tasks is allocated more budget and the server with low-criticality tasks is allocated a lower budget. Since suspend-resume of tasks is already an implicit feature of HSF, this approach avoids the overheads of handling individual tasks by managing the budget of servers. Using servers of an HSF implicitly ensures that a temporal fault of a lower criticality task does not impact the higher criticality tasks.
- Another approach is to keep a heterogeneous set of tasks in the server and keep the criticality associated with tasks. This way the current design and implementation can be kept as is and it can be investigated how to use multiple plug-ins in conjunction with ExSched, as ExSched currently supports only a single plug-in.

The fact that now one has servers that require varying amounts of execution time at unpredictable moments, schedulability is an issue here. In the first option, the criticality level is a global property, so the schedule can be verified per criticality level. This is hard, if not impossible in the second scheme. It requires determining schedulability for each combination of criticality levels of the various servers and so might actually lose the advantage of MC to use less bandwidth than required when absolute (CA) WCETs.

Besides this, Mode Changes can also be explored and be related to criticality level changes.

Bibliography

- [1] "Automotive Safety Integrity Level." [Online]. Available: http://en.wikipedia.org/wiki/Automotive_Safety_Integrity_Level
- [2] "Fault tolerance." [Online]. Available: http://en.wikipedia.org/wiki/Fault_tolerance
- [3] "Hardware virtualization." [Online]. Available: http://en.wikipedia.org/wiki/Hardware_virtualization
- [4] "Latex." [Online]. Available: <http://en.wikibooks.org/wiki/LaTeX>
- [5] "Robustness (computer science)." [Online]. Available: [http://en.wikipedia.org/wiki/Robustness_\(computer_science\)](http://en.wikipedia.org/wiki/Robustness_(computer_science))
- [6] "Ubuntu kernel repository." [Online]. Available: <http://kernel.ubuntu.com/~kernel-ppa/mainline/>
- [7] M. Åsberg, "Synthesis and synchronization support for hierarchically scheduled real-time systems," Ph.D. dissertation, Mälardalens Högskola Eskilstuna Västerås, 2014.
- [8] M. Åsberg, T. Nolte, S. Kato, and R. Rajkumar, "ExSched: An External CPU Scheduler Framework for Real-Time Systems," in *18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA12)*, August 2012, pp. 240 – 249. [Online]. Available: <http://www.es.mdh.se/publications/2564->
- [9] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, pp. 284–292, 1993.
- [10] R. Barry, *Using the FreeRTOS Real Time Kernel: PIC32 Edition*. Real TimeEngineers, 2011. [Online]. Available: <http://books.google.nl/books?id=c9ontwAACAAJ>
- [11] S. Baruah and A. Burns, "Implementing mixed criticality systems in ada," in *Proceedings of the 16th Ada-Europe International Conference on Reliable Software Technologies*, ser. Ada-Europe'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 174–188. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2018027.2018045>
- [12] S. Baruah and S. Vestal, "Schedulability analysis of sporadic tasks with multiple criticality specifications," in *Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, ser. ECRTS '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 147–155. [Online]. Available: <http://dx.doi.org/10.1109/ECRTS.2008.26>
- [13] S. K. Baruah, A. Burns, and R. I. Davis, "Response-time analysis for mixed criticality systems," in *Proceedings of the 2011 IEEE 32Nd Real-Time Systems Symposium*, ser. RTSS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 34 – 43. [Online]. Available: <http://dx.doi.org/10.1109/RTSS.2011.12>
- [14] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. J. Bril, "Towards Hierarchical Scheduling on top of VxWorks," in *4th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT08)*, July 2008, pp. 63 – 72. [Online]. Available: <http://www.es.mdh.se/publications/1261->

- [15] A. Burns and S. Baruah, "Towards a more practical model for mixed criticality systems," in *Proc. WMC, RTSS*, 2013, pp. 1–6.
- [16] A. Burns and R. Davis, "Mixed Criticality Systems - A Review," 2013. [Online]. Available: www-users.cs.york.ac.uk/~burns/review.pdf
- [17] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, "LITMUS^{RT}: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers," in *IEEE 27th International Real-Time Systems Symposium*, December 2006, pp. 111 – 126.
- [18] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.
- [19] G. M. de A. Lima and A. Burns, "An Optimal Fixed-Priority Assignment Algorithm for Supporting Fault-Tolerant Hard Real-Time Systems," in *IEEE Transactions on Computers*. IEEE, October 2003, pp. 1332 – 1346.
- [20] Z. Deng and J. W. s. Liu, "Scheduling real-time applications in an open environment," in *in Proceedings of the 18th IEEE Real-Time Systems Symposium, IEEE Computer*. Society Press, 1997, pp. 308–319.
- [21] P. Graydon and I. Bate, "Safety Assurance Driven Problem Formulation for Mixed-Criticality Scheduling," in *Proceedings of the 1st International Workshop on Mixed Criticality Systems*, December 2013, pp. 19 – 24.
- [22] M. Holenderski, R. J. Bril, and J. J. Lukkien, *Real-Time Systems, Architecture, Scheduling, and Application*. InTech, April 2012, ch. An Efficient Hierarchical Scheduling Framework for the Automotive Domain, pp. 67 – 94.
- [23] M. Holenderski, W. Cools, R. J. Bril, and J. J. Lukkien, "Extending an open-source real-time operating system with hierarchical scheduling," Technische Universiteit Eindhoven, Tech. Rep., October 2010. [Online]. Available: <http://alexandria.tue.nl/repository/books/692166.pdf>
- [24] M. J. Holenderski, M. van den Heuvel, R. J. Bril, and J. Lukkien, "Grasp: Tracing, visualizing and measuring the behavior of real-time systems," in *1st WATERS*, July 2010. [Online]. Available: <http://www.win.tue.nl/san/grasp/>
- [25] R. Inam, J. Mäki-Turja, M. Sjödin, and M. Behnam, "Hard Real-time Support for Hierarchical Scheduling in FreeRTOS," in *7th annual workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 11)*, July 2011, pp. 51–60, visited on: 2014.03.13. [Online]. Available: <http://www.es.mdh.se/publications/2166->
- [26] S. Kato, R. Rajkumar, and Y. Ishikawa, "A loadable real-time scheduler suite for multicore platforms," Technical Report CMU-ECE-TR09-12, Tech. Rep, Tech. Rep., 2009.
- [27] B. Krosse, "Interoperable GCDc AutoMation Experience," October 2013. [Online]. Available: <http://www.gcdc.net/>
- [28] J. J. Labrosse, *Microc/OS-II*, 2nd ed. R & D Books, 1998.
- [29] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari, "Aquosa: Adaptive quality of service architecture," *Software: Practice and Experience*, vol. 39, pp. 1 – 31, April 2008. [Online]. Available: <http://retis.sssup.it/~tommaso/publications/SPE-2008.pdf>
- [30] J. Regehr and J. A. Stankovic, "HLS: A Framework for Composing Soft Real-Time Schedulers," in *IEEE 22nd Real-Time Systems Symposium*. IEEE, December 2001, pp. 3 – 14.

- [31] F. Santy, L. George, P. Thierry, and J. Goossens, "Relaxing mixed-criticality scheduling strictness for task sets scheduled with fp," in *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, ser. ECRTS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 155–165. [Online]. Available: <http://dx.doi.org/10.1109/ECRTS.2012.39>
- [32] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees." in *IEEE 24th International Real-Time System Symposium(RTSS)*. IEEE, December 2003, pp. 2 – 13.
- [33] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, ser. RTSS '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 239–243. [Online]. Available: <http://dx.doi.org/10.1109/RTSS.2007.35>
- [34] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1 – 36:53, May 2008. [Online]. Available: <http://doi.acm.org/10.1145/1347375.1347389>

Appendices

Appendix A

Hierarchical Scheduling Framework

This chapter discusses the various options for real-time schedulers that support HSFs. A choice for a scheduling framework will be made based on the requirements. In chapter 4, it has been mentioned that core requirement is Linux as an operating system running on a desktop grade processor. The requirement of Linux as an operating system rules out the option of using an HSF developed for other operating systems like FreeRTOS [25], VxWorks [14], $\mu\text{C}/\text{OS-II}$ [23] and Windows 2000 [30]. The design and development of a new HSF is out of scope of the project. Hence, following three options of an HSF developed for Linux are available:

- ExSched [8]
- LITMUS^{RT} [17]
- AQuoSA [29]

These frameworks will be analysed on the following criteria.

- Availability of the Framework
- Availability of source code
- Support for multi-core schedulers
- OS version independence

Table A.1 presents a comparison of the frameworks based on the criteria mentioned above.

Table A.1: HSF Comparison

Criterion	ExSched	LITMUS ^{RT}	AQuoSA
Availability of the Framework	Y	Y	Y
Availability of source code	Y	Y	Y
Support for multi-core schedulers	Y	Y	N
OS version independence	Y	N(patch based)	N(patch based)
Comments	Independent of the version of the Vanilla Linux kernel	Focuses on multi-core scheduling	-

LITMUS^{RT} is a patch-based implementation which is a deterrent as in due course of i-Game another version of Linux kernel may be chosen. Although LITMUS^{RT} supports multi-core scheduling which would be important (since later in the project we will focus on multi-core scheduling), ExSched offers the same benefits without requiring patches to the vanilla kernel.

AQuoSA is a strong competitor of ExSched in the fact that it offers two features, namely adaptive Quality of Service (QoS) and feedback control, which are very beneficial for the first step of the assignment. These features are the result of the Resource Reservation layer that is capable of dynamically adapting the CPU allocation for QoS aware applications based on their run-time requirements. However, it lacks multi-core scheduling. Adding multi-core scheduling support to AQuoSA implies additional work to be performed for i-Game. Also AQuoSA is a patch-based implementation and has the same disadvantage as LITMUS^{RT} if another version of the Linux kernel is chosen. AQuoSA and ExSched both offer the feature of plug-in development as extensions for the framework.

In contrast, ExSched lacks the adaptive Quality of Service and feedback control features offered by AQuoSA. ExSched is developed for the Vanilla versions of the Linux kernel. It is also independent of any patches, hence, it can be easily ported to any mainline Linux kernel. ExSched supports multi-core scheduler development as well and offers pre-built multi-core schedulers as plug-ins. The multi-core schedulers implemented in ExSched framework are:

- Global scheduling
 - G-FP(Global-Fixed Priority)
 - FP-US
- Partitioned scheduling
 - FP-FF(Fixed Priority, First Fit heuristic)
- Semi-Partitioned scheduling
 - FP-PM

The portability benefits and availability of multi-core schedulers in ExSched overcome the temporary advantages of using AQuoSA. Response time analysis is not in the scope of the PDEng assignment but it would be essential within the scope of i-Game. ExSched uses a real-time scheduling class, i.e., `rt_sched_class`, to isolate real-time tasks from non-real-time tasks. Non-real-time tasks are scheduled by a fair scheduling class, i.e., `fair_sched_class`. It is also possible to use ExSched with the PREEMPT_RT patch if further isolation and low latency is required.

In this report, we use ExSched on the Linux Vanilla kernel version 2.6.36 and an Intel Core I5 processor. The processor used is quad-core but we will boot Linux with parameter (MAX_CPUS=1) such that Linux uses only one core.

Appendix B

Implementation Set-up

In this chapter, we discuss the set-up done to implement the scheme. We discuss the set-up of the operating system in section B.1. Then we discuss the set-up of extended ExSched in section B.2 and the tool used for analysis of the results in section B.3.

B.1 Operating System Set-up

The kernel version 2.6.36 was downloaded from [6]. The process of installation is following:

1. Extract the files downloaded from the repository in an empty directory. The extracted files have the ".deb" extension.
2. Make sure there are no other files in the directory than the extracted files.
3. Open a terminal and navigate to the directory that contains the extracted ".deb" files.
4. Execute the following command in the terminal

```
sudo dpkg -i *.deb
```

5. Wait for the installation to complete
6. When the installation is complete, the grub needs to be updated. Execute the following command:

```
sudo update-grub
```

7. Reboot the system
8. Now two kernels are installed. The system will automatically boot the default version selected for the Linux kernel.
9. If the boot menu screen does not show at the system start-up, press and hold the "shift" key. Now you will be able to see the two kernel versions installed in the menu.
10. Select the kernel version 2.6.36 (generic) to boot Linux.
11. To validate if Linux is booted with the correct kernel version, execute the following command:

```
uname -r
```

For making Linux use only one core follow the following procedure:

1. Open the grub configuration file at location "/boot/grub/grub.conf" in an editor.

2. Search for the line with "linux /boot/vmlinuz-2.6.36-020636-generic".
3. Append "maxcpus=1" to the line.
4. Save and reboot the system. Linux should now use only one processor.
5. To validate if the system is using only one core execute the following command:

```
nproc
```

For further validation use the following command:

```
lscpu
```

Using this command you will see the number of cores in the CPU.

Alternatively you can edit the boot options from the boot menu.

1. While the system is booting press and hold the "shift" key.
2. When the boot menu screen is displayed, highlight (do not select) the option with kernel version 2.6.36 and press the "e" key.
3. Search for the line with "linux /boot/vmlinuz-2.6.36-020636-generic".
4. Append "maxcpus=1" to the line.
5. Press "b" key to boot the system.

For controlling the clock frequency follow the following steps:

1. Open a terminal
2. Install the 'cpufrequtils' using the command
3. Execute the following command which will set the minimum and maximum frequencies to the highest. The following command will set the frequencies only for the first core.

```
sudo apt-get install cpufrequtils
```

```
sudo cpufreq-set -c 0 -g performance
```

4. Execute the following command to ensure that setting the frequencies was successful.

```
cpufreq-info
```

B.2 Extended ExSched Set-up

Although the set-up procedure of ExSched is mentioned in its documentation for the sake of this report we repeat the set-up procedure. The procedure for the extended version of ExSched with support for monitoring and mixed criticality schemes is the same. The set-up process is following:

1. Download the framework from "<http://www.idt.mdh.se/~exsched/>" for Linux.
2. Extract the downloaded file. A folder "Linux" will be created.
3. Open a terminal and type 'uname -r' in the command shell and you will see your kernel release string < RELEASE_STRING >
4. Create the directory path '/usr/src/kernels/< RELEASE_STRING >/include/resch'

5. In the terminal navigate to the "Linux/core" directory.

6. Execute the following commands:



```
make clean # this will clean all executables present
chmod 777 configure # to make the file executable
./configure # setup the config.h header file with variables
cd script # navigate to script subdirectory of the Linux\core
            folder
chmod 777 install # to make the file executable
chmod 777 uninstall # to make the file executable
cd .. # navigate to the core subdirectory of the Linux folder
make # compile the RESCH kernel module
sudo make install # install the kernel module and
cd ../library # navigate to the library subdirectory of the Linux
            folder
make clean # this will clean the compiled file if present
make # compile the library
cd ../mysample # navigate to the mysample subdirectory of the Linux
            folder
make clean # this will clean the compiled file if present
chmod 777 start.sh # to make the file executable
cd trace # navigate to the trace subdirectory of the Linux/mysample
            folder
chmod 777 convert.awk # to make the file executable
chmod 777 ftraceToGrasp.sh # to make the file executable
chmod 777 grasp # to make the file executable
cd .. # navigate to the mysample subdirectory of the Linux folder
make # compile the tasks
./start.sh # run the tasks with the provided parameters
```

In the mysample subdirectory the tasks are defined which are compiled and run by the start.sh script. Using the steps mentioned above the tasks will run until completion. Although ExSched allows infinite periodic job releases for experimental purposes, the tasks are allowed to release only six jobs. The execution time of some jobs is purposely larger than its allowed WCET. This will allow us to validate our implementation and generate results for analysis.

B.3 Grasp Tool Set-up

A version of grasp tool is present in the ExSched set-up. In the "Linux\mysample\trace" folder the grasp executable is present. One can download the latest executable of grasp from [24]. The "ftraceToGrasp.sh" script generates a file named "out.txt" in the "Linux\mysample\trace" folder. The "out.txt" file has timing logs for the task's execution. The grasp tool will create the visualization-graph using the timing logs in the "out.txt" file. When the tasks have finished execution, use the following commands to generate graphs for analysis:

```
cd trace # navigate to trace subdirectory of Linux/mysample folder
./ftraceToGrasp.sh # it will dump the timing logs in the out.txt file
                    and launch the grasp tool with the out.txt file as a command line
                    parameter
```

Wait for some time until grasp tool generates the timing graph. Use the  button on the top-left corner to scale down the graph or  to scale-up.