



Flexible Task Management for Self-Adaptation of Mixed-Criticality Systems with an Automotive Example

Daniel Andreas Krefft

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr.rer.nat.)

genehmigten Dissertation.

Vorsitzende:

Prof. Dr. Claudia Eckert

Prüfende der Dissertation:

Prof. Dr. Uwe Baumgarten

Prof. Dr.-Ing. Andreas Herkersdorf

Die Dissertation wurde am 22.08.2018 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 11.12.2018 angenommen.

Abstract

With regard to future connected cars, there are two trends leading to consolidated hardware devices as well as an increasing software complexity within a car. In consequence, a rising number of software needs to use the provided resources of a few high-performance hardware devices. For an efficient resource usage, a flexible software management supporting the (self-)adaptation of a software system is getting more and more important - even within a car. This flexible software management therefore needs to consider the criticality and real-time properties of an application within this context. Corresponding to the hardware consolidation, the management approach should be combined with the actual application on one hardware device. With a recent advance of multi-core embedded systems, there exists a potential hardware platform which is able to support this combination in an embedded context. Especially, the usage of a flexible management supporting the self-adaptation of a mixed-criticality software system on an embedded hardware device during run-time is of interest.

For integrating a flexible software management on top of an embedded hardware platform, this work investigates the application of a controlled adaptation process considering the given resources. The work, thus, describes a consolidated architecture approach which designs the flexible software management as part of an operating system. The corresponding adaptation process is based on a newly designed run-time integration framework which allows the deployment of mixed-criticality applications into a given software system. For realizing the several decision phases within the integration framework, an extended description considering the static and dynamic information of an application and the system itself is described. With the usage of a multi-core platform, the decision making processes for the adaptation can be executed beside the actual applications on the same device.

The implementation is done by using a micro-kernel based real-time operating system. As part of a distributed environment, the implemented system is tested according to certain automotive scenarios. Furthermore, the system is also tested in isolation through a software-in-the-loop comparable approach. The first evaluation results present an indication if the demanded resource and adaptation requirements are fulfilled. Moreover, this work presents a fundamental feasibility study for adapting an automotive system during operation.

In conclusion, this thesis explores aspects for integrating a flexible software management approach in an embedded mixed-critical system on a multi-core hardware platform and demonstrates the feasibility of adapting such a system during operation.

Zusammenfassung

Mit Hinblick auf zukünftige vernetzte Fahrzeuge, gibt es zwei Trends, welche sowohl zu konsolidierten Steuergeräten, als auch zu einer zunehmenden Komplexität von Software im Fahrzeug führen werden. In der Konsequenz muss eine steigende Anzahl von Software die Ressourcen von wenigen leistungsstarken Steuergeräten verwenden. Für eine effiziente Nutzung dieser Ressourcen wird eine flexible Verwaltung zur Unterstützung der (Selbst-)Adaption eines Software Systems immer wichtiger - sogar im Fahrzeug. Diese flexible Softwareverwaltung muss die Kritikalitäts- und Echtzeiteigenschaften einer Anwendung in diesem Kontext berücksichtigen. Entsprechend der Hardwarekonsolidierung sollte die Verwaltung mit der eigentlichen Anwendung auf einem Gerät ausgeführt werden. Mit den jüngsten Entwicklungen von Mehrkernprozessoren in eingebetteten Systemen existiert eine mögliche Plattform welche diese Kombination unterstützen kann. Besonders die Nutzung einer flexiblen Verwaltung, welche die Selbst-Adaption im Bereich gemischt-kritischer Software Systeme auf einem eingebetteten Gerät zur Laufzeit unterstützt, ist hierbei von Interesse.

Für die Integration einer flexiblen Softwareverwaltung untersucht diese Arbeit die Anwendbarkeit einer kontrollierten Adaption eines Software Systems unter Berücksichtigung der gegebenen Ressourcen. Die Arbeit beschreibt hierzu eine konsolidierte Architektur, welche die flexible Softwareverwaltung als Teil eines Betriebssystems vorsieht. Der Adaptionprozess selbst basiert auf einem, in dieser Arbeit entwickelten, Laufzeit-Framework zur Integration von gemischt-kritischen Anwendungen in ein gegebenes Software System. Zur Realisierung der einzelnen Entscheidungsprozesse innerhalb des Frameworks wird eine erweiterte Beschreibung zur Erfassung der statischen und dynamischen Informationen über die Anwendung und das System selber präsentiert. Durch die Nutzung eines Mehrkernprozessors können die Entscheidungsprozesse für die Adaption neben den eigentlichen Anwendungen auf dem selben Gerät ausgeführt werden.

Die Umsetzung des vorgestellten Ansatzes erfolgt unter Verwendung eines microkernel-basierten Echtzeitbetriebssystems. Das System wird in verschiedenen automotiven Anwendungsszenarien als Teil eines verteilten Bordnetzes beziehungsweise, isoliert, in einem Software-in-the-Loop Ansatz getestet. Die ersten Ergebnisse geben eine Einschätzung darüber, inwieweit die geforderten Aspekte zur Nutzung der verfügbaren Ressourcen und zum zeitlichen Verhalten der Adaption erreicht werden konnten. Darüber hinaus präsentiert diese Arbeit eine grundsätzliche Machbarkeitsstudie über die Möglichkeit das automotive System mit Hilfe des entwickelten Ansatzes zur Laufzeit zu adaptieren.

Zusammenfassend untersucht diese Arbeit die Aspekte zur Integration einer flexiblen Softwareverwaltung in einem eingebetteten gemischt-kritischen System auf einer Mehrkernprozessor-basierten Hardware-Plattform und zeigt die Machbarkeit solch ein System während der Ausführung zur Laufzeit zu adaptieren.

Acknowledgments

First of all, I would like to thank Prof. Dr. Uwe Baumgarten for his dedication giving me the possibility to work at his chair of operating systems and for the opportunity to work on such an interesting thesis topic. His door has always been open for conversations, discussions and giving me the right directions. Further, I would like to thank my mentor Prof. Dr. Andreas Herkersdorf for giving me valuable remarks and for his readiness to examine my thesis.

Many thanks to all my colleagues at the chair of operating systems. I met many special people who inspired me in our pro-active meetings, talks and day-to-day work. Especially Susanne Guggenmos and Sebastian Eckl provided me a huge support. Many thanks also to all students contributing to the work we developed during my time at the chair, including the creation of our prototype, where Alexander Reisner and Alexander Weidinger showed their enthusiasm and dedication.

I am deeply thankful to my family and my friends who supported me with open ears and good advice over the years. Special thanks go to Woody Lemcke for proofreading my thesis.

I would like to thank my parents, my mother Belinda Krefft and my father Raimund Krefft. This thesis would not have been possible without all their love and support also in difficult times.

Last but not least, a great thank you goes to a special person in my life. With your smile, love and enthusiasm you have given me the motivation for this long run. Thank you, Susanne Mozes.

Contents

Abstract	iii
Zusammenfassung	v
Acknowledgments	vii
Contents	ix
List of Figures	xiii
List of Tables	xv
Acronyms	xvii
1 Introduction	1
1.1 Background and Current Trends	1
1.2 Motivation	3
1.3 Problem and Contributions	6
1.4 Structure of the Work	8
2 Fundamentals of Embedded Systems Development	9
2.1 Basic Taxonomy of a System	9
2.2 Architecture in Software Engineering	10
2.3 Functional and Design Requirements	13
2.3.1 Taxonomy of Self-Adaptation	14
2.3.2 Taxonomy of Dependability	16
2.4 System Analysis Foundations of an Embedded System	21
2.4.1 System Architecture	22
2.4.2 Real-time Scheduling	26
2.4.3 Communication	33
3 Domain Analysis	35
3.1 Operating System Architectures for Multi-Core Systems	35
3.2 Real-Time Scheduling of Dynamic Mixed-Critical Systems	38
3.3 Mixed-Critical System Properties	42
3.4 Self-Adaptive Systems Properties	44
3.5 Differences between Related Work and Proposed Approach	47

4	Design of a Consolidated Self-Adaptive Software Architecture for Multicore Systems	53
4.1	Description of the Overall System Architecture	53
4.2	Kernel Space and User Space Interactions	56
4.3	Allocation of Software Components to Multicore Hardware Platform . . .	60
4.3.1	Integration of Application Components within the System	61
4.3.2	Criticality-Aware Allocation of Software Components	63
4.3.3	Isolation Supporting Criticality	66
4.4	Description of Workflows for Self-Adaptive Software Architecture Changes	68
4.4.1	Adding a New Task into the System	69
4.4.2	Optimizing current System State	70
4.4.3	Monitoring current System State	71
5	Flexible Task Management	75
5.1	Basic Model for Reasoning about Current Running State	75
5.1.1	Ready-Queue and Core Representation within User Space	75
5.1.2	Extended Task Model Considering Criticality and Schedulability .	76
5.2	Run-Time Task Integration Framework	77
5.2.1	Critical-Aware Dispatching of Tasks to Cores	78
5.2.2	Short-Term Online Admission Test	81
5.2.3	Long-term Knowledge-based Optimizer	84
5.2.4	Synchronizing User Space with Kernel Space	86
5.3	Combine the Framework Concept with the Designed Architecture	88
5.3.1	Co-Existent Scheduling Strategies	89
5.3.2	Tracing/Logging Software Component	89
5.3.3	Controller Software Component	90
5.3.4	Synchronizer Software Component	91
6	Extension of a Microkernel-Based Operating System	93
6.1	Genode and Fiasco.OC Basic Concepts	93
6.1.1	L4 Fiasco.OC Microkernel	94
6.1.2	Genode Operating System Framework	96
6.2	Extension of the Fiasco.OC Microkernel	99
6.2.1	Adding additional Scheduling Policies	100
6.2.2	Extension of Time-related Thread Information	102
6.2.3	Extension of Scheduling-Context for Thread Deployment and In- formation Gathering	103
6.3	Operating System Framework Components	105
6.3.1	Component for Network Communication	105
6.3.2	Components for Information Gathering	107
6.3.3	Component for Controlling the System	110
6.3.4	Components for Loading and Deploying of Tasks	114
6.4	Limitations of the current Implementation	116
6.4.1	Getting System Information	116

6.4.2	Adding New Tasks to the System	118
7	Evaluation of the Flexible Task Management	121
7.1	Scenario I: Reliable Autonomous Driving under Adaptation	121
7.1.1	Installing an Application Component During Operation	121
7.1.2	Updating an Application Component During Operation	124
7.2	Scenario II: Testing System Properties using Artificially Generated Task Sets	126
7.2.1	Support for the Separation of Software Components	127
7.2.2	Support for the Avoidance of Overload Situations	131
7.2.3	Support for Update Induced Dynamic Task Set Changes	133
8	Conclusion	137
9	Future Work	141
A	Code Listings	143
A.1	Source Code	143
A.2	Acceptance Test in Pseudocode	155
B	Evaluation Setup	159
B.1	Generating Artificial Task Sets	159
B.2	Autonomous Driving Setup	163
B.2.1	Hybrid Test Bed	163
B.2.2	Installation Scenario Additional Files	165
	Own Publications	169
	Advised Theses	171
	Bibliography	173

List of Figures

1.1	Fields of application for connected-car-solutions (cited after Statista [116])	2
1.2	ECU consolidation foresees the usage of smaller but more powerful hardware devices [21]	2
1.3	Software innovations fuel automotive and mobility advances but quickly increase complexity [48].	3
2.1	Architecture and design link requirements and code together [82]	11
2.2	AUTOSAR process to configure a system [9]	12
2.3	Generic Observer/Controller Architecture [107]	13
2.4	Overview about Self-Adaptation and their Instances	15
2.5	Taxonomy of Dependability presented as a tree [10]	16
2.6	Faults, errors and failures according to Kopetz [88]	16
2.7	ISO26262 Process Diagram Overview [75]	19
2.8	SEooC Development Lifecycle (cited after [131])	22
2.9	Overview of an CPS system [41]	23
2.10	Simplified Architecture of a Multicore Processor	24
2.11	SMP Processor Configuration with Operating System	25
2.12	Taxonomy of real-time scheduling algorithms [88]	26
2.13	Summary of times associated with a task execution [95]	28
2.14	Scheme of the guarantee mechanism used in dynamic real-time systems (according to [29])	30
2.15	Kopetz 10.2: Necessary and sufficient schedulability test	31
2.16	Scheduling comparison	32
2.17	Car2Car protocol stack (cited after [138])	34
4.1	Block Diagram of the Overall Task Management Architecture	54
4.2	Mandatory components and their connectors	57
4.4	Allocation Example following the Partition Process	67
4.6	Activity diagram for inserting a new task within the system	70
4.7	Activity diagram for optimizing a ready queue	71
4.8	Activity diagram for diagnosing the current system state	72
5.1	Flowchart describing the allocation of newly arriving LO-tasks to a respective core [113]	80
5.2	Overall Synthesis Process [108]	82
5.3	Sufficient and Exact Test in the context of other software components [53]	84
5.4	Abstract system overview with tracing/logging, controller, and synchronization [113]	88

List of Figures

5.5	Abstract overview of the optimization process of the controller software component [113]	90
6.1	Overall Overview about Fiasco.OC and Genode	93
6.2	A ready-queue with threads containing their scheduling parameter is assigned to each core. A scheduler populates this ready queue. [68]	95
6.3	Genode inter-process communication [64]	99
6.4	Collaboration Diagram for Fiasco.OC Microkernel	100
6.5	Simplified flow of Fiasco.OC and Genode interface	102
6.6	Simplified flow of Fiasco.OC and Genode interface for Deploy	104
6.7	Collaboration Diagram of several classes for the flexible task management in Genode	106
6.8	Collaboration diagram for information gathering (i.e. tracing)	108
6.9	Collaboration diagram for Sched_controller	111
6.10	Flow Chart for Optimization Procedure according to [112]	115
6.11	Genode and Fiasco.OC priority values [64]	118
7.1	Installation Case Setup Car and Workstation	122
7.3	Update Case Setup Car and Workstation	124
7.4	Update of component on <i>Core1</i> over time	126
7.5	Plot shows a periodically executed task on <i>Core1</i> using a timing service on <i>Core0</i>	128
7.6	Plot shows a task calculating PI on <i>Core1</i> without using any services on <i>Core0</i>	130
7.7	Plot shows a task calculating PI on <i>Core1</i> with management components running on <i>Core0</i>	132
7.8	Histogram View of optimizing process	134
7.9	Piechart of average timing demands of several execution steps	135
7.10	Piechart of average timing demands of several steps during execution	135
7.11	Timing measurements of different applications	136
B.1	Detailed toolchain view between offline and online system	159
B.2	Schematic View of Evaluation Setup	164
B.3	Test Bed of Evaluation Setup	164
B.4	Physical model car equipped with boards	165

List of Tables

2.1	ASIL Assignment according to ISO26262	21
5.1	Comparison of schedulability analyses for FP [53]	83
5.2	Comparison of schedulability analyses for EDF [53]	83
5.3	Comparison of lock-based and lock-free mechanism [66]	87
5.4	Comparison between Mutex, RCU and STM [33]	87
6.1	Scheduler Operation	96
6.2	magic codes provided by the network component	107
B.1	Generated Tasks and their descriptions (idp js/mz)	162

Acronyms

AC Autonomic Computing.

ACC Adaptive Cruise Controller.

AMP Asymmetric Multi-Processing.

ASIL Automotive Safety Integrity Level.

AUTOSAR AUTomotive Open System ARchitecture.

BSW AUTOSAR Basic Software.

CAN Controller Area Network.

CBS Constant Bandwidth Server.

COTS Custom Off The Shelf.

CPS Cyber-physical System.

CPU Central Processing Unit.

ECU Electronic Control Unit.

EDF Earliest Deadline First.

FP Fixed Priority.

GUI Graphical User Interface.

HARA Hazard Analysis and Risk Assessment.

IEC International Electrotechnical Commission.

IEEE Institute of Electrical and Electronics Engineers.

IPC Inter Process Communication.

ISO International Organization for Standardization.

LCT Latest Completion Time.

Acronyms

LITMUS^{RT} Linux Testbed for Multiprocessor Scheduling in Real-Time Systems.

MMU Memory Management Unit.

MUF Maximum Urgency First.

OC Organic Computing.

OS Operating System.

OSADL Open Source Automation Development Lab.

OSEK Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen.

OSI Open Systems Interconnection.

RAM Random-Access Memory.

RCU Read Copy Update.

RM Rate Monotonic.

RPC Remote Procedure Call.

RTA Response Time Analysis.

RTE Run-Time Environment.

RTOS Real-Time Operating System.

SMP Symmetric Multi-Processing.

STM Software Transactional Memory.

SW-C AUTOSAR Software Component.

VBS Value-Based Scheduling.

WCET Worst Case Execution Time.

XML Extensible Markup Language.

1 Introduction

This chapter will give an introduction to the greater context of this work. An overview about current automotive trends with their supposed implications for the development of future embedded systems in this area will be outlined in section 1.1. The subsequent section 1.2 will illustrate the motivation for this work which is derived from the former implications. With the given motivation, possible challenges on the way towards a solution which addresses the motivated aspects will be explained in section 1.3. After that, certain contributions which are made by this doctoral thesis will be outlined. The chapter will close with the overall structure of the work in section 1.4.

1.1 Background and Current Trends

In January 2016, McKinsey & Company published a report about possible perspectives of technology-driven trends for the automotive industry towards 2030 [102]. One of the eight key perspectives on the “2030 automotive revolution” considers the changes driven by connectivity services and feature upgrades. This outlook is supported by a current study [60] which estimates that the number of connected cars (i.e. predecessor of autonomous driving cars) will increase globally up to 220 million by 2020. This development is accompanied by a rising number of new application fields for connected-car-solutions (see figure 1.1). A majority of these applications and services are mainly driven by pure software-based solutions. Unlike before, certain services need software maintenance mechanisms (e.g. evaluate, control, and modify) directly in the car. In case of an automatic software update for instance, (i.e. “diagnosis & maintenance”) a car’s software can be altered during operation. The resource demands (e.g. computation power) of these new services can also be seen as a driving force for future trends in the area of automotive hardware development.

One clear trend which is identifiable in the area of automotive (embedded) hardware (see figure 1.2) is the development of smaller and simultaneously more powerful devices (e.g. domain controllers). This development is mandatory due to the fact that the computation effort which is needed by the new services will potentially increase. In a first step, the performance gain was achieved by increasing the clock speed of a single processor. This approach has reached its limits in recent years because of a negative impact regarding simplicity (i.e. getting more complex), efficiency and power consumption. In order to still allow further performance gains, recent embedded hardware is designed to use multiple computation units on the same device. This is achievable by using a smaller structural size allowing a tighter packing of electronic computing elements on hardware units (i.e. constant hardware construction size). This development enables the practical

1 Introduction

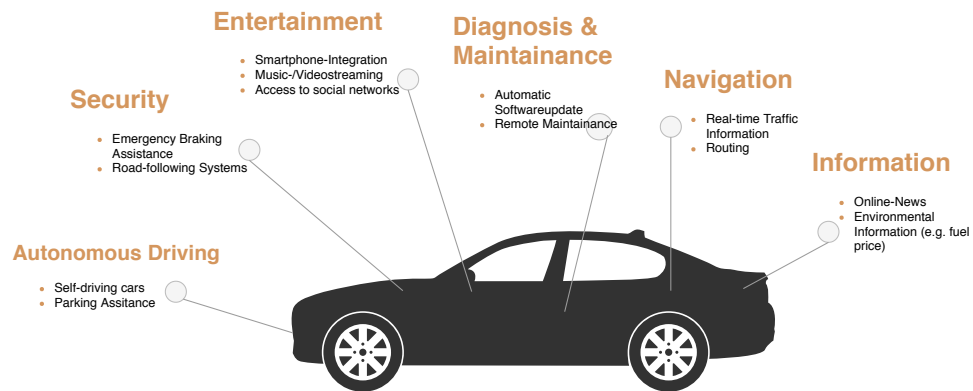


Figure 1.1: Fields of application for connected-car-solutions (cited after Statista [116])

design of multi-core embedded systems [105]. The trend is definitely going towards embedded multi-processor and multi-core hardware which allows an independent execution of software on distinct cores but on the same device. Now it is possible to consolidate software functionality on one device rather than using several devices. For example, it is possible to integrate maintenance mechanism on one core beside cores which are providing the actual functionality. In this case, an automatic software update can be prepared on the maintenance core without disturbing the execution of the other cores. With this trend in the area of embedded hardware, there are also new possibilities to manage and use software within a car.

ECU Consolidation

Into a distributed central compute platform

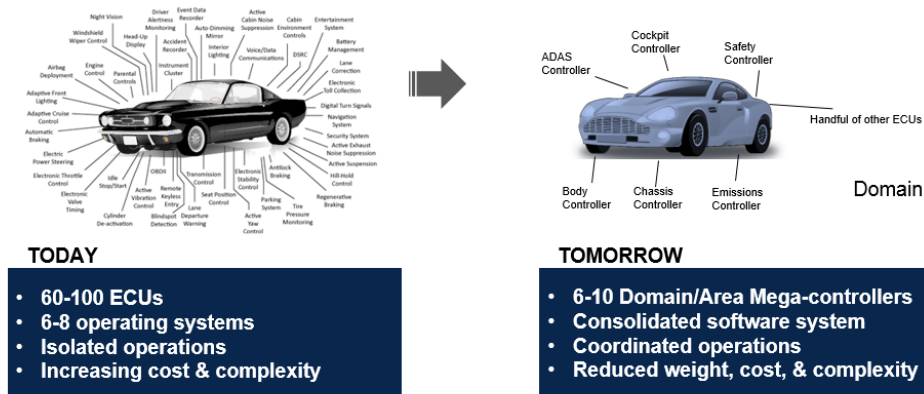


Figure 1.2: ECU consolidation foresees the usage of smaller but more powerful hardware devices [21]

As shown in figure 1.3, there has been an increasing number of software features within a car over the last years. The great majority of these new features are belong to the domain of driving assistance functions with the goal of autonomous driving cars in the foreseeable future. Today, the current capabilities of a car's driving assistance functionality already exceeds the actual control and regulation tasks to drive a car (e.g. X-by-wire). This will be enforced with the advance of future autonomously driving cars which will be steadily merged with their environment. Software is, hence, one of the main driving factors of future cars and it's getting more and more complex. Adequate handling of (complex) software inside a car is therefore of growing importance [24]. Especially, supporting the flexible (i.e. dynamic at run-time) adaptation of a car's system software depending on its environment needs to be mastered. If a well known software maintenance mechanism would be available in a car, the required system software adaptation would be feasible.

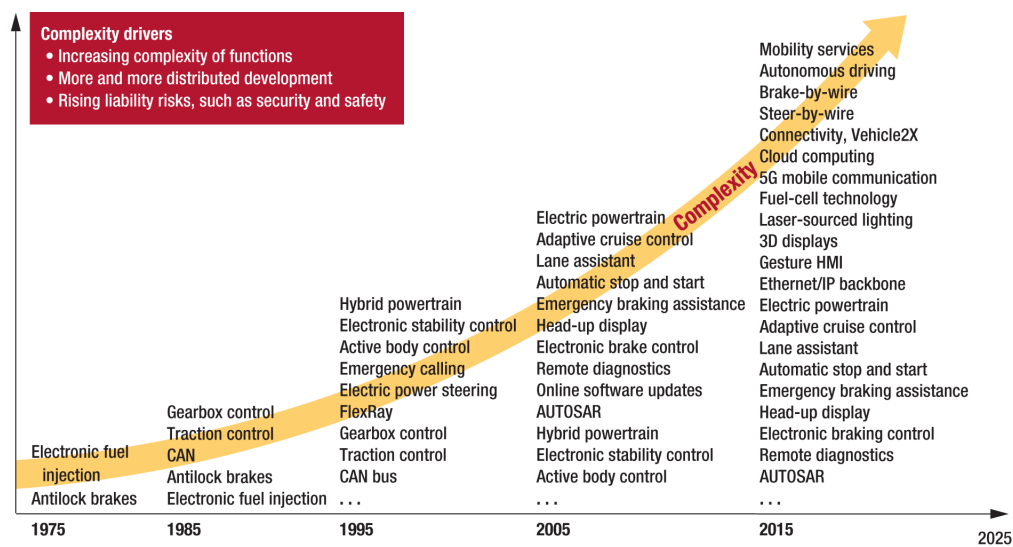


Figure 1.3: Software innovations fuel automotive and mobility advances but quickly increase complexity [48].

In conclusion, due to a raising software complexity within automotive systems there is a common desire to update already delivered system software. With a recent advance of multi-core embedded systems, there exists potential hardware platforms which are able to supply an update mechanism even in an embedded context. The integration of software updates however is challenging due to several constraints that software within an automotive system needs to consider.

1.2 Motivation

The trend of a rising software complexity within a car could already be observed in other areas not related to automotive industry. In general, several domains suffer from

1 Introduction

an increasing complexity of software systems where software, and its development, is the driving factor. This problem always is subject to the area of software engineering, where methods and approaches are developed to control the complexity of a system. In other domains, outside of an automotive context, there are already methods and approaches to control the complexity of a system. One of the established approaches is to design a flexible system which is allowed to adapt its inner workings to a given situation. For the context of this work this means to control the rising complexity of software-based services within an embedded system by finding adequate software management mechanisms. The overall focus relies here on the mechanisms which are proven to work well in other domains (e.g. autonomic computing) but are not immediately available in the area of embedded systems. Especially, the usage of dynamic software adaptation mechanisms in embedded systems is of interest. Automotive functions however are used for demonstrating the flexibility of the resulting architecture achieved through software management. The main motivation is to apply software management techniques to a category of devices whose software was not intended to be updated. The motivation in this section therefore outlines the potential benefits but also the challenges which can arise by using these mechanisms.

Most of flexible run-time software management approaches require that the system can be modified during operation. Due to the fact that automotive systems are highly integrated (mixed-critical) systems, there exists an integration challenge for such approaches in general [126]. In contrast to, for instance, server software and their related adaptation mechanism, a modification of embedded software components within a car during run-time is more limited. The severity in case of a failure during an adaptation process of an automotive embedded system software differs greatly from a comparable process in a data center. The resource constraints within an embedded system are stronger in comparison to a data center system. The time when a reaction (i.e. behavior change) needs to take place is another critical factor. In the case of a car as part of a traffic situation, an excessive reaction time can lead to an accident. This situation leads to a broad utilization of real-time operating systems within an automotive embedded system because the correct system behavior does not solely depend on the correct function execution but also on the time when this function gets executed.

In the area of real-time (operating) systems, the realization of a flexible adaptation mechanism which dynamically manages the resources during run-time can be seen as a challenge. In general, the design of a real-time system to operate as expected is subject of certain requirements [88]. These requirements mainly concern the limits and conditions (i.e. dependability and timing) under which a real-time system is able to provide its functionality. In more detail, the programs which are executed on a real-time system may have different timing or safety conditions. An integration of those programs on one system is, in case of safety, considered as a mixed criticality system. In this case, an embedded real-time system consists of several programs with varying importance for the overall system and its dependability. As a result, the possibility of adding extra software features to a mixed-critical embedded system during operation is not consistent with the safety requirements of those systems. To be compliant with these limitations, the development approach for designing a mixed criticality systems is focused on design-

time rather than run-time. Current embedded systems therefore are designed to be immutably during run-time.

Due to the fact that most embedded systems are designed to make the best fit to their current task, current deployment strategies for embedded systems exclude the possibility to update the system during operation at run-time. For the execution of its main task, in most cases an embedded system basically does not need to be modified at run-time. A closer look onto the interaction between an embedded system and its environment offers the system's main task for measuring and controlling the contextual physical processes and their dynamics by orchestrating actions that influence the processes [95]. An embedded system, hereby, does not necessarily change its internal working structure nor its behavior. Future (automotive) trends however are leading to a dramatic increase of automation and to a rising influence of these systems by their surrounding environment. To overcome these future challenges, many current embedded systems (i.e. interface function between physical and virtual world) need to be transformed towards cyber-physical systems.

There are several situations in which an embedded system could benefit from an adaptation of its software. If a scenario was not foreseeable during the design process and therefore not tested, an embedded system could fail to manage this new situation during run-time. Scenarios where such an uncertainty can arise is most likely in cases of autonomous driving where a car is moving through a changing environment. With the capability of adapting the software afterwards, these situation could be managed. A further example is the retrofitting of software features to handle new standards (e.g. 5G communication protocol). Opening an embedded system in the form of a connected car to its environment entails the risk for software attacks which could be circumvented by patching the affected parts.

In conclusion, flexible software management in embedded systems is also a resource management problem. The provided limitations of an embedded system need to be respected. The enhancement of a current embedded system with a dynamic adaptation mechanism requires combining flexibility and integrity in one single system. The used resources therefore need to be partitioned between the applications and the software management. This partition needs to be guaranteed by an adequate separation mechanism. In embedded real-time systems, all software programs either static or dynamic have an important aspect in common, namely time. A real-time system ensures the correct system behavior by guaranteeing timing conditions of its software. Timing aspects, thus, are the key to real-time systems. In combination with mixed-critical systems, the main task of an embedded real-time system is to ensure the timing conditions according to each software's criticality level and to prevent their mutual interference by accessing the same hardware resource. A consistent resource management that is applicable to a great variety of use-cases can be provided by a real-time operating system. There is one software component in an operating system which is a basic mechanism for the management of time (i.e. assigning CPU resources to software programs) - namely a scheduler. Supporting a flexible software management, the underlying scheduling mechanism needs to be able to manage modifications of its set of to be scheduled actions, unlike current approaches in embedded real-time systems. Nevertheless, the run-time

1 Introduction

adaptation needs to respect the conditions and limits of an embedded system for guaranteeing a stable system behavior. If an arbitrary adaptation for instance would be possible, software may enter unforeseeable states that have never been tested or reasoned about at design-time possibly leading to incorrect system behavior. The scope of a system's adaptation behavior therefore needs to be limited by run-time control mechanisms (supporting the actual scheduling). Basically, the intended control mechanisms are inspired by the current design-time steps for developing an embedded system. As a result, the controlling tasks include the monitoring, diagnosing, correcting, and adapting of system's software [36]. For a correct system behavior, a (mobile) embedded system needs to autonomously execute these steps in a changing environment. Without the possibility of a human controller, an embedded system needs to be controlled by itself (self-x). As a result, flexible software management will be enabled by distinct operating system support via the scheduler.

1.3 Problem and Contributions

Within this section, the central research questions will be identified. Furthermore, the contributions of this work will be enumerated.

Research Questions:

The central research problem within the context of this work is the software-based adaptation of an embedded mixed-critical system during operation.

Resource Partitioning With the possibility of altering existing applications within a system, a demand arises where applications need to be correctly integrated or deployed into the system during adaptation. Within the context of a mixed-critical system, different applications acquire different resources of the system. Considering the required resource partitioning in combination with a possible deployment of applications, the following questions for a concrete partitioning scheme arise:

- *Research Question 1:* how could a separation of adaptation management and applications look like?
- *Research Question 2:* how could a critical-aware resource allocation of an application during run-time look like?

Adaptation Control An uncontrolled adaptation of applications within a mixed-critical system during operation can be hazardous for the overall system state. Dedicated software components and mechanisms for controlling the adaptation are therefore required. There are several questions related to this.

- *Research Question 3:* how could an adaptation control on top of an operating system scheduler look like by keeping a small trusted computing base?

- *Research Question 4*: how could a concrete adaptation process look like that supports dynamic mixed-critical workloads?
- *Research Question 5*: how could a concrete control mechanism look like that is suitable for achieving a robust system execution?

Resource Usage and Overhead In the context of an embedded system, there are strict resource constraints. The mechanism used for the adaptation of the system therefore needs to be resource efficient. A central question, thus, is:

- *Research Question 6*: which timing behavior does the adaptation management and overall process show?

Contributions:

The following contributions will be presented in this work.

1. overall contributions
 - a) adaptation management and application execution is located on the same device
 - b) an adaptation of the system takes place during operation at run-time
 - c) required separation of timing resources is achieved by utilizing distinct cores on a multi-core platform via a co-existent (partitioned) scheduling scheme
2. architecture contributions
 - a) design of an architecture to enable a separation between applications and the adaptation management as well as a separation between high critical and low critical applications
 - b) integrating the adaptation management within the operating system user-space of the designed architecture
3. process contributions
 - a) identification of concrete workflows for the adaptation process
 - b) presentation of kernel ready queues as well as an extended application model within user-space
 - c) design of an user-space run-time integration framework for controlling the adaptation process
4. implementation contributions
 - a) extension of an existing capability-based microkernel to provide required information, methods and scheduling strategies for the adaptation management
 - b) implementation of relevant software components in user-space via the Genode Operating System Framework

1 Introduction

5. evaluation contributions

- a) software-based only separation on a multi-core platform will be presented
- b) coexistent scheduling strategies managing the underlying cores will be presented
- c) a possible self-optimizing behavior for a robust system execution in spite of changing conditions (i.e. continuous installation of applications) will be presented
- d) several timing measurement of distinct adaptation stages as well as the overall adaptation process will be presented
- e) a case study of a live update/installation within an automotive driving scenario (hybrid simulator test bed) during operation will be presented

1.4 Structure of the Work

The remainder of this thesis is organized in the following chapters. Chapter 2 gives a fundamental introduction of thesis relevant topics according to the area of cyber-physical systems, scheduling theory and adaptive systems. In the following, chapter 3 provides an analysis of existing solutions in the area of flexible task management. Therefore, this chapter provides information about current system approaches for flexible task management as well as related work. After that, the derived requirements summarize the identified corner points for the realization of a flexible task management. The chapter concludes with the assumptions and restrictions taken for this thesis. After that, chapter 4 describes the overall architecture in more detail and inspects each element of the resulting architecture regarding its functionality, interface and behavior. Additionally, it provides a detailed explanation about the taken deployment decisions of software to hardware platform and the description of controlling mechanisms for an adaptive behavior. In chapter 5, the intended concept of a flexible task management will be outlined. First, a task model will be described which serves as base for relevant analysis. Followed by the concept of a run-time integration framework for the management of variable task sets, the chapter concludes with the combination of the run-time integration framework into the designed architecture. The implementation of the proposed architecture and flexible task management will be shown in chapter 6. In detail, the chapter shows the extension of a given microkernel-based operating system with the propagated design changes and their relevance for the implementation. The chapter 7 covers the executed test cases for the implemented approach as well as their implications considering the overhead and efficiency. The last chapters 8 and 9 close this thesis with a critical rating of the undergoing research and its findings as well as open topics for future research. Appendix A and appendix B contain the source code listings and a detailed description about the used test setup.

2 Fundamentals of Embedded Systems Development

As outlined in chapter 1, this work concerns flexible software management in the area of embedded systems. This chapter is therefore structured according to the development process of an embedded system and introduces, thereby, relevant terms for this work. Beginning with section 2.1, the definition of a system as well as related terms (e.g. architecture) will be given. After that, section 2.2 outlines the general development process for embedded systems and addresses the role of an architecture within this process. The greater area of software requirements and especially design requirements will be covered in section 2.3. The closing section 2.4 will provide an overview about a core task of the design process, namely the system analysis. Relevant terms according to system and software architecture (i.e. operating system, multi-core platform, real-time scheduling, and communication) will be in the focus.

2.1 Basic Taxonomy of a System

At a first sight, the area of embedded systems has induced a vast number of varying devices which are ubiquitous in our daily life. Despite their different applications, common software engineering methods are used for the development and design process of embedded systems. This section therefore introduces basic terms from the area of systems and software engineering which are not necessarily restricted to embedded systems at all.

A system can be commonly understood as a set of entities which are related among each other. The entities could be arbitrary. In the context of this work, a system itself could also be an entity as part of a greater system which interacts with other entities. This view of a system corresponds to the definition by Avizienis [10]:

Definition 1 *A **system** is an entity that interacts with other entities, i.e., other systems, including hardware, software, humans, and the physical world with its natural phenomena.*

According to this definition an embedded system can be seen as a combination of a hardware system and a software system. A hardware system is composed of a set of hardware resources like memory, processor, bus and so on. A software system is correspondingly composed of a set of software elements like functions, modules, components, objects etc..

Later on, in section 2.3 two types of requirements will be introduced, namely functional requirements and design requirements. In advance, a functional requirement describes

what a system should perform in case of a certain event. These expectations can be defined as a systems function:

Definition 2 *The **function** of a system is what the system is intended to do and is described by the functional specification in terms of functionality and performance [10].*

Functions define what is expected from a system but without providing a concrete solution which is actually defined by the behavior of a system.

Definition 3 *The **behavior** of a system is what the system does to implement its function and is described by a sequence of states [10].*

The so far introduced taxonomy of a system describes solely a set of entities and gives an indication that these entities are linked together. The taxonomy however lacks the definition about the way these entities are related to each other. The term of a structure will be defined as follows.

Definition 4 *The **structure** of a system describes, the way how the entities of a system are linked together.*

The structure thus describes how the system is organized inside. Another term which is related to the structure and the internal organization is the definition of a system's architecture.

Definition 5 *A system **architecture** is the highest structural level describing the fundamental organization of a system's entities (i.e. their relationships among others) and the principles governing a system's design and evolution.*

Due to the fact that entities of a system could also be systems (i.e. with their own architecture), an architecture describes the structure of a system as hierarchy. The aspect of system evolution in this definition describes that an architecture doesn't merely describe the static structure of a system but also dynamic changes. In general, an architecture exists on an abstract level above algorithms or concrete data types. A software architecture can therefore be seen as the highest level of a greater software system [134]. This applies to a hardware architecture also.

In conclusion, the basic taxonomy of a system was introduced in this section. Several definitions therefore were given. In the context of this thesis, software management is of interest. The following taxonomy therefore is focusing on a software engineering process rather than a general system engineering process. In a next step, the role of an architecture within a software engineering process will be given.

2.2 Architecture in Software Engineering

Software engineering can be seen as a process where several methods are used to develop software in a structured and reproducible way. Considering current procedure models (e.g. v-model, waterfall), the overall goal of this process is to subsequently transform initial system requirements to an actual executable system (i.e. source code).

Starting with a system analysis. A software system which is described by its requirements is analyzed (i.e. striped down). The corresponding requirements serve hereby as an input for this process. After that, the relations between the system entities are formally described. A software architecture is hereby the output of the system analysis. Requirements and code thus are interlinked by an architecture [82]. The architecture presents a suitable assignment of requirements to architecture elements. The following design poses principles for the realization of the architecture elements in an implementable form. This fact is again depicted in figure 2.1

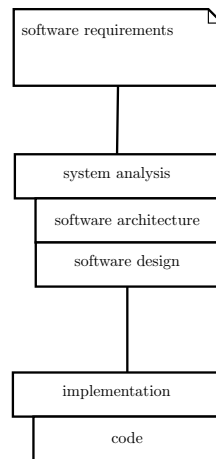


Figure 2.1: Architecture and design link requirements and code together [82]

This section will describe the role of an architecture within a software engineering process. An architecture combines the requirements and the resulting code. The second part of this section therefore shows two concrete examples of architectures with different approaches for designing a system.

AUTOSAR Architecture

In the following, the AUTOSAR standard [40] should serve as an example of a model-based approach for the development of automotive software systems. AUTOSAR stands for AUTomotive Open System ARchitecture and defines the following key aspects:

- Architecture which is based on a layered architecture model which can be departed in Application, RTE and Basic Software
- Methodology is used to enable a deployment of software across ECU-boundaries as well as the configuration of basic software of each ECU
- Application Interfaces defines interfaces of typical automotive software applications to enable a later integration

Beside other aspects of the development process, the AUTOSAR methodology describes an important step (see figure 2.2) to configure a system (i.e. “Configure System”).

According to the AUTOSAR standard, an application is modeled as a composition of interconnected components. Where each component has well-defined “ports”, through which the component can interact with other components. During this configuration step the components are mapped on specific system resources (ECUs). This deployment is assisted by a tooling support which automates a part of the process. Thereby, the virtual connections between the components are mapped onto local connections (within a single ECU) or on network-technology specific communication mechanisms (such as CAN or FlexRay frames).

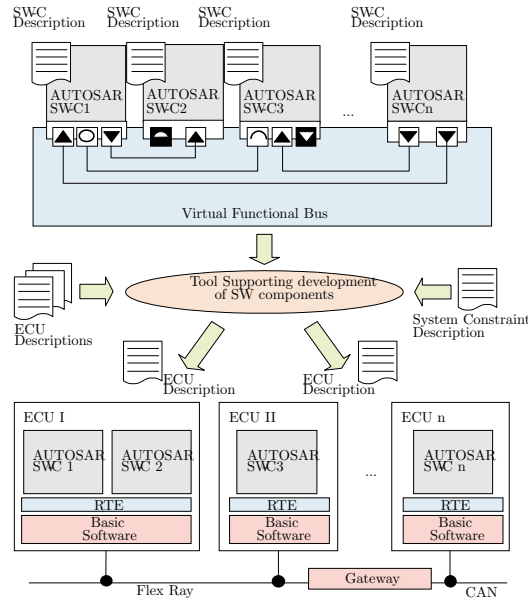


Figure 2.2: AUTOSAR process to configure a system [9]

The described process for designing and implementing an automotive system via the AUTOSAR methodology is by far not the only way to develop modern embedded systems. The next section covers a further architecture where an other approach will be pursued.

Organic Computing Architecture

In case of a traditional engineering paradigm, there is a central methodology (i.e. process) which guides through the required steps for developing an embedded system (cp. safety process in section 2.3.2). This paradigm is mostly anchored in the design-time of the development process. Another approach however allows the system to evolve during operation. In contrast to the engineering paradigm, in the evolutionary paradigm there is no centralized or coordinated planning [107]. Where in classical engineering the behavior of a system (i.e. states) is completely described, the course of evolution is, by definition, not determined by describing a given state of the evolving unit [107]. This has several implications for the design of an architecture considering the evolutionary paradigm.

First, there is no complete methodology which can be used to process from requirements to the final implementation [46]. Also, there is no clear way to process design requirements (e.g. self-x properties) to functional requirements (cp. safety process). In general, there is no process standard for the development of self-x capable systems compared to other embedded systems (like automotive). Despite the lack of a complete methodology, there are tools, paradigms, and references which can be used to develop the systems in mind. An important paradigm for developing an architecture of a system with self-x properties is the generic observer/controller architecture (shown in figure 2.3).

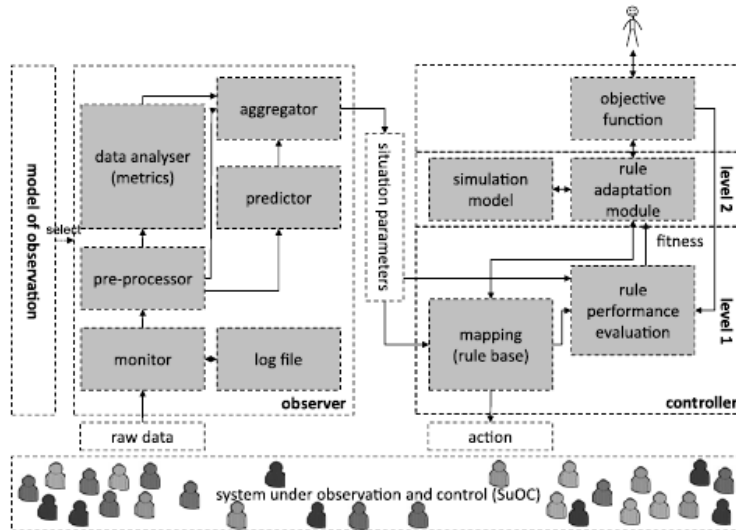


Figure 2.3: Generic Observer/Controller Architecture [107]

With an observer/controller architecture it can be determined which software components are needed to support self-x properties in the targeting architecture. The concrete design of these components however is left to the developer. There is no process which takes the requirement of self-adaptation and results in distinct software components which are equipped to realize the desired behavior. The task of a monitoring component, for instance, is defined but it lacks a methodology which can be used to generate functional requirements, derive an architecture and implement a component with exactly these monitoring properties. Designing a system which evolves over time may require this freedom during the development process. An architecture for evolving systems thus represents a paradigm which can be used as starting point for own system analysis.

2.3 Functional and Design Requirements

At the beginning of a software engineering process there are requirements which specify, in terms of functions, what a system should do in case of a distinct event. There are basically two types of requirements defined in the context of software engineering: functional requirements and design (quality) requirements.

Definition 6 A *functional requirement* defines what should be done by the system in case of a certain event. It therefore defines the functionality of a system.

Functional requirements specify [82]:

- the qualification for the purpose
- interoperability (co-working with other systems)
- security (access control)

Design requirements in contrast can be seen as requirements where a direct solution, as with functional requirements, is not derivable. A requirement like “the ACC functionality needs to conform with an ASIL D classification” needs a further detailing step before a concrete realization can be made. Behind a design requirement there are also functional requirements. Design requirements thus are not finished functional requirements.

The *design requirements* specify (selection) [82]:

- efficiency (timing and resource usage)
- maintainability
- safety (dependability)
- portability

Definition 7 A *design requirement* specifies what is demanded from a system in terms of quality.

These design requirements thus determine how the architecture needs to be designed. Safety, for instance, requires the design of distinct functionality to conform with standards (e.g. ISO26262) which means that a certain failure probability needs to be met. Guaranteeing these requirements forms the implementation process as well as the design of a functionality. So, design requirements formulate what is required but not how it is achieved. Because of the high relevance of this thesis (i.e. dealing with software management in embedded mixed-critical system), design requirements will be discussed in more detail in subsequent subsections. To specify all demands on software quality however both functional and design requirements are needed.

2.3.1 Taxonomy of Self-Adaptation

This section will cover the maintainability of a system through self-x properties. The overall classification schema of self-adaptation can be summarized as depicted in figure 2.4. Hereby, a self-adaptation property of a system can be divided into structural and behavioral modifications. Possible structural modifications are reconfiguration, compositional adaptation and recomposition.

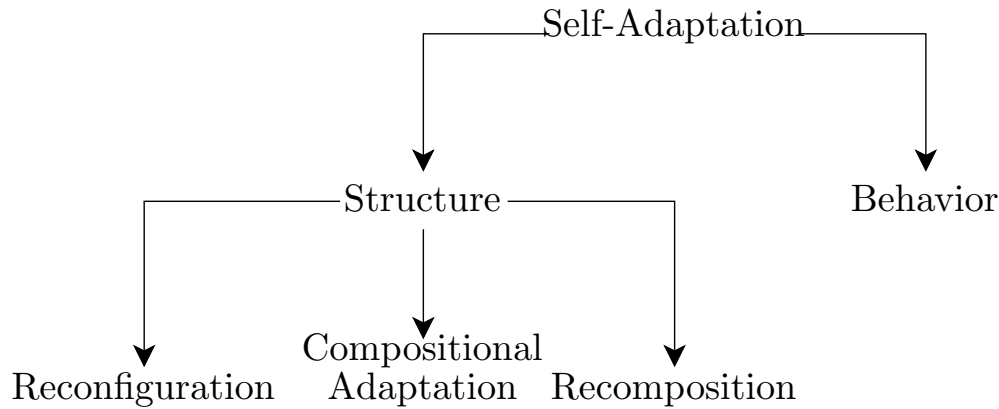


Figure 2.4: Overview about Self-Adaptation and their Instances

Definition 8 A software entity has the capability of **self-adaptation** if it can automatically change its structure or behavior, based on observations of its system or environment, with the goal to maintain or improve the Quality of Service [121].

Definition 9 A **compositional adaptation** changes the structure by adding or removing of software components.

Definition 10 A **behavioral adaptation** changes the behavior (control flow or quality of service) of a software component without directly affecting component execution sequences [121]. The behavioral adaptation thus concentrates on the adaptation of service parameters.

Definition 11 A **recomposition** changes the allocation from software components and resources (e.g. migration from one control unit to another).

After defining the different types of self-adaptation, there are several types when an adaptation can be performed. Considering the work of Rohr [121], three different activation types can be distinguished.

Definition 12 A system can be typed as **reactive** if the adaptation only happens after a certain threshold was achieved (e.g. performance degradation).

Definition 13 A system can be typed as **predictive** if the adaptation will be executed before a certain event occurs.

Definition 14 A system can be typed as **proactive** or **goal-directed** if the system has a normal execution but decides for itself to gain a better state (e.g. performance increase).

In this section, the taxonomy of self-x system properties considering the self-adaptation of a system were defined. The next section will give an overview about the taxonomy of dependability.

2.3.2 Taxonomy of Dependability

Further design requirements which are relevant for this thesis are targeting the dependability which is mandatory for a real-time system. The notion of dependability covers the design attributes of a system that are relating to the quality of service a system delivers to its users during an extended interval of time. The taxonomy used in this section is inspired by the work of Avizienis [10] where a taxonomy overview is depicted in figure 2.5.

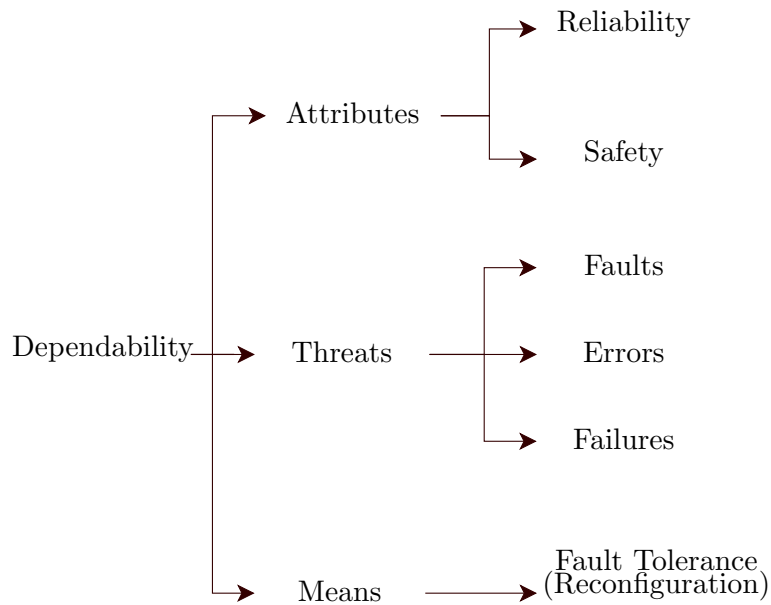


Figure 2.5: Taxonomy of Dependability presented as a tree [10]

Beginning with the threats of the dependability, this section introduces the general terms of fault, error and failure. The general relation of these terms is depicted in figure 2.6. A fault, hereby, is the cause of an error which in turn causes a deviation of a intended service resulting in a failure.

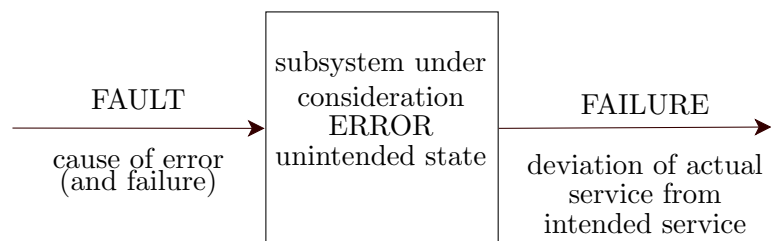


Figure 2.6: Faults, errors and failures according to Kopetz [88]

For further investigations the term of fault and failure are relevant. With accordance to a system service, Avizienis [10] defines a failure as follows:

Definition 15 *Correct service is delivered when the service implements the system function. A **service failure**, often abbreviated here to **failure**, is an event that occurs when the delivered service deviates from correct service [10].*

In relation to a service failure an error and a fault can be defined as:

Definition 16 *Since a service is a sequence of the system's external states, a service failure means that at least one (or more) external state of the system deviates from the correct service state. The deviation is called an **error**. The adjudged or hypothesized cause of an error is called a **fault** [10].*

Avizienis further distinguishes two levels of failures considering the relation between the benefit provided by a service without a failure and the consequences of failures:

- **minor failures**, where the harmful consequences are of similar cost to the benefits provided by correct service delivery;
- **catastrophic failures**, where the cost of harmful consequences is orders of magnitude, or even incommensurately, higher than the benefit provided by correct service delivery (cp. criticality).

Especially the level of catastrophic failures gives a first hint to the definition of criticality (see below). As a result, the term of a fail-safe system can be defined as follows:

Definition 17 *A system whose failures are, to an acceptable extent, all minor ones is a **fail-safe** system [10]. A fail-safe system has been defined as a system, where the application can be put into a safe state in case of a failure. At present, the majority of industrial systems that are safety-relevant fall into this category [88].*

An interesting remark is the fact that, according to Kopetz [88], a great number of today's industrial systems are fail-safe. This observation harmonizes with the fault-tolerance methods provided in the context of self-x properties in the below text. In a next step the attributes of dependability will be outlined.

There are several attributes for the system dependability where reliability and safety are the most important inside the context of this thesis. Later on, the term of robustness will be added.

To start with the reliability, Kopetz [88] defines reliability as follows:

Definition 18 *The **Reliability** $R(t)$ of a system is the probability that a system will provide the specified service until time t , given that the system was operational at the beginning, i.e., $t = t_0$.*

In accordance to provide the taxonomy of safety, the former definition of a catastrophic failure can be extended to the term of criticality:

Definition 19 ***Criticality** is a designation of the level of assurance against failure needed for a system component [26].*

Considering safety, Kopetz [88] defines it as:

Definition 20 *Safety can be defined as the probability that a system will survive a given time-span without the occurrence of a critical failure mode that can lead to catastrophic consequences [88]. In other words, Safety is reliability regarding critical failures.*

Where the combination of both terms lead to the definition of safety-critical systems. Here the bridge to the timing demands of a hard real-time system is obvious:

Definition 21 *A computer system becomes **safety-critical** (or hard real-time) when a failure of the computer system can have catastrophic consequences, such as the loss of life, extensive property damage, or a disastrous damage to the environment [88].*

Where another definition is more relevant from the point of real-time system and timing demands itself:

Definition 22 *Informally, a **safety critical real-time system** can be defined as one in which the damage incurred by a missed deadline is greater than any possible **value** that can be obtained by correct and timely computation. A system can be defined to be a hard real-time system if the damage has the potential to be catastrophic.[8]*

In conclusion, a service failure (e.g. caused by a missed deadline) of a safety critical (hard real-time) system leads to catastrophic consequences. But there are systems which are not exclusively safety critical. More and more systems provide functions with different criticalities:

Definition 23 *A **mixed-criticality architecture** can be understood as structural system description, where applications of different criticality can coexist in a single integrated architecture and the probability of any unintended interference, both in the domains of value and time, among these different-criticality applications must be excluded by architectural mechanisms [88].*

For the means to attain the dependability, Avizienis [10] defines fault prevention, fault tolerance, fault removal and fault forecasting. He concludes that a combination of all four means is mandatory. For this thesis however the focus relies on fault tolerance which avoids failures by error detection and system recovery [10]. Moreover, the fault handling via reconfiguration fits to the concept of self-x system properties which was provided in section 2.3.1.

Process Standard for Functional Safety

This section will give an example of a safety standard which describes the process to transform design requirements to functional requirements. This standard is known as ISO26262. According to ISO (the International Organization for Standardization), ISO 26262 is the adaptation of IEC 61508 to address the sector specific needs of electrical and/or electronic (E/E) systems within road vehicles. ISO 26262 includes guidance to mitigate the risks from systematic failures and random hardware failures by providing a reference for the automotive safety life cycle (depicted in figure 2.7). and supports

2.3 Functional and Design Requirements

the tailoring of the activities to be performed during the life cycle phases, i.e., development, production, operation, service, and decommissioning. Furthermore, the standard provides an automotive-specific risk-based approach to determine integrity levels (Automotive Safety Integrity Levels (ASIL)) which are used to specify which of the requirements of ISO 26262 are applicable to avoid unreasonable residual risk. The remainder of this section provides selected definitions (i.e. relevant for this thesis) from the extensive standard work [75].

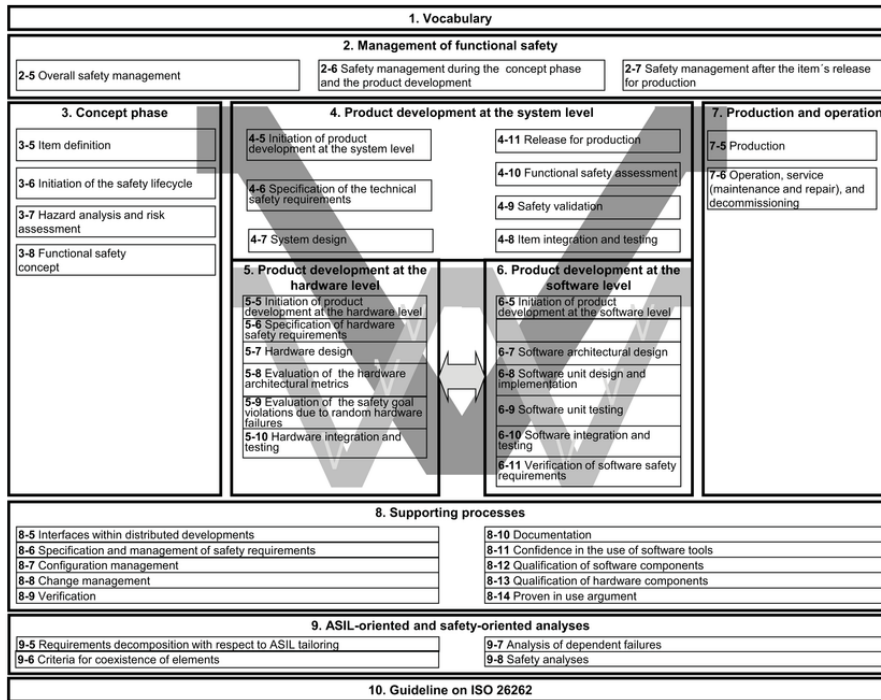


Figure 2.7: ISO26262 Process Diagram Overview [75]

An important step for the beginning of the safety process is the hazard analysis (step 3.7 in figure 2.7). This analysis is commonly defined as:

Definition 24 *Hazard Analysis and Risk Assessment (for short: HARA) method to identify and categorize hazardous events of items and to specify safety goals and ASILs related to the prevention or mitigation of the associated hazards in order to avoid unreasonable risk*

The hazard analysis therefore outputs an assignment of items to their corresponding integrity level. An Automotive Safety Integrity Level is hereby defined as:

Definition 25 *Automotive Safety Integrity Level ASIL is one of four levels to specify the item's or element's necessary requirements of ISO 26262 and safety measures to apply for avoiding an unreasonable risk, with D representing the most stringent and A the least stringent level.*

2 Fundamentals of Embedded Systems Development

Where the term risk can be seen as the result of the following equation (according to [49]): $risk = severity \times (exposure \times controllability)$, according to the definition of ISO.

Definition 26 A *risk* is a combination of the probability of harm and the severity of that harm.

The related terms severity, exposure and controllability give information about the extent, the frequency, and the ability to avoid a specific harm:

Definition 27 *severity* estimate of the extent of harm to one or more individuals that can occur in a potentially hazardous event

Definition 28 *exposure* state of being in an operational situation that can be hazardous if coincident with the failure mode under analysis

Definition 29 *controllability* ability to avoid a specified harm or damage through the timely reactions of the persons involved, possibly with support from external measures

These attributes are classified by a consistent schema:

- Class of Severity (S)
 - S0: No injuries
 - S1: Light and moderate injuries
 - S2: Severe and life-threatening injuries (survival probable)
 - S3: Life-threatening injuries (survival uncertain), fatal injuries
- Class of probability of exposure regarding (E)
 - E0: Incredible
 - E1: Very low probability
 - E2: Low probability
 - E3: Medium probability
 - E4: High probability
- Class of controllability (C)
 - C0: Controllable in general
 - C1: Simply controllable
 - C2: Normally controllable
 - C3: Difficult to control or uncontrollable

		C1	C2	C3
S1	E1	QoS	QoS	QoS
	E2	QoS	QoS	QoS
	E3	QoS	QoS	A
	E4	QoS	A	B
S2	E1	QoS	QoS	QoS
	E2	QoS	QoS	A
	E3	QoS	A	B
	E4	A	B	C
S3	E1	QoS	QoS	A
	E2	QoS	A	B
	E3	A	B	C
	E4	B	C	D

Table 2.1: ASIL Assignment according to ISO26262

Based on this classification, corresponding levels will be selected depending on the risk according to table 2.1:

The resulting safety classification is the first step on the way from design requirements to functional requirements. In the next step this classification is used to form concrete functional requirements. The approach during development of AUTOSAR, for instance, is comparable to a Safety Element out of Context (SEooC) approach as described in ISO DIS 26262-10, chapter 10. The SEooC approach is that safety goals or safety requirements of the targeted element (e.g. a software unit) are replaced by assumptions (see figure 2.8). These assumptions (e.g. failure modes to be detected and handled in this software unit) are the basis for the implementation of such a generic software element [131]. The lowest right box in figure 2.8 represents the software requirements to be implemented either as software-component (SW-C) or as a feature in a specific basic software (BSW). This approach is an example demonstrating the complete way from defining and analyzing of design requirements and their systematic refinement to functional requirements which can be implemented.

2.4 System Analysis Foundations of an Embedded System

The so far discussed requirements are independent from a later realization. An architecture however combines requirements and implementation. For this, a knowledge about the available resource is required to quantify if a requirement (e.g. efficiency) is fulfilled. Furthermore, the management of such resources is of interest. This section therefore has covered the area of system analysis within a software engineering process in more detail. As a result, a software architecture as well as a software design will be provided. The process of a system analysis for an embedded system however also requires a minimal understanding of a system architecture which combines the software architecture and a hardware platform. Both system architecture and software architecture therefore will be

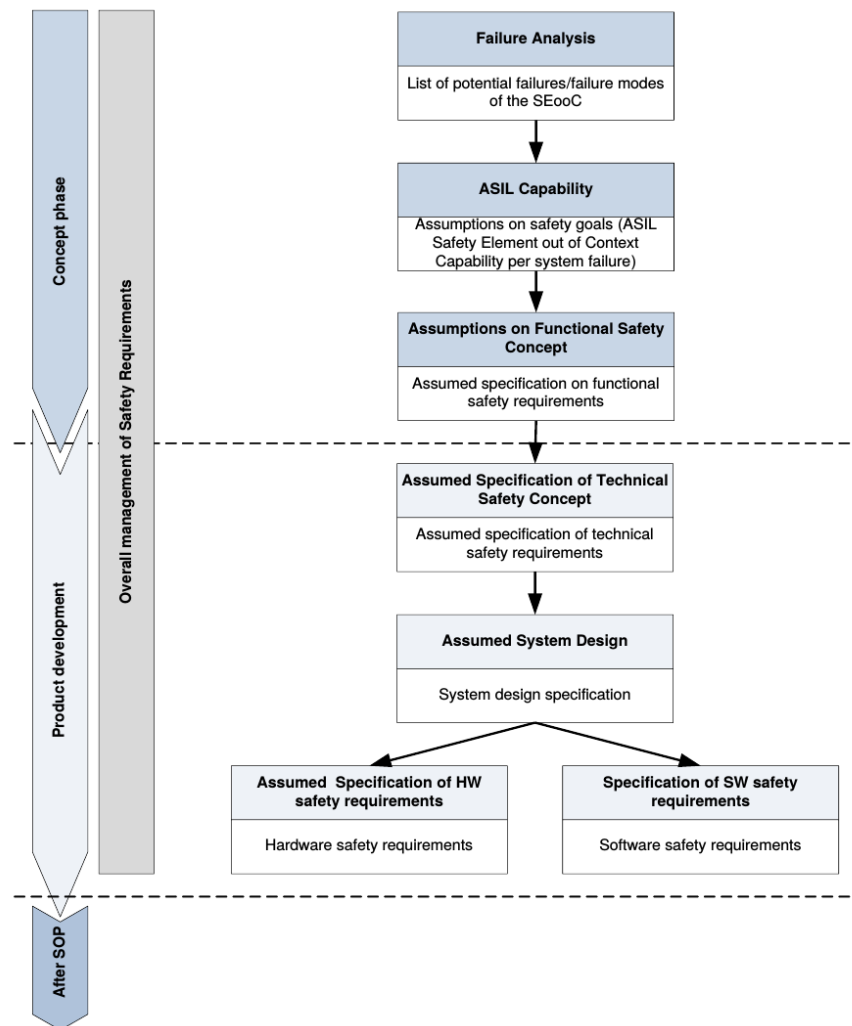


Figure 2.8: SEooC Development Lifecycle (cited after [131])

described in subsection 2.4.1 respectively. After that, subsection 2.4.2 covers the topic of scheduling for real-time systems. This section closes with the most fundamental terms concerning communication (see section 2.4.3).

2.4.1 System Architecture

A software system is generally executed on top of some sort of hardware system. Embedded systems are no exception. This section therefore exemplifies a cyber-physical system (see figure 2.9) to consider the available resources which need to be managed by the software system. Due to the fact that scheduling is an integral part of the later

software management, the focus relies on the computing resources (i.e. processors). In this context, a cyber-physical system is defined as:

Definition 30 *Embedded Real-time system and Cyber-Physical System.*

A **cyber-physical system** (CPS) is an integration of computation with physical processes whose behavior is defined by both cyber and physical parts of the system. Embedded computers and networks monitor and control the physical processes, usually with feedback loops where physical processes affect computations and vice versa [95].

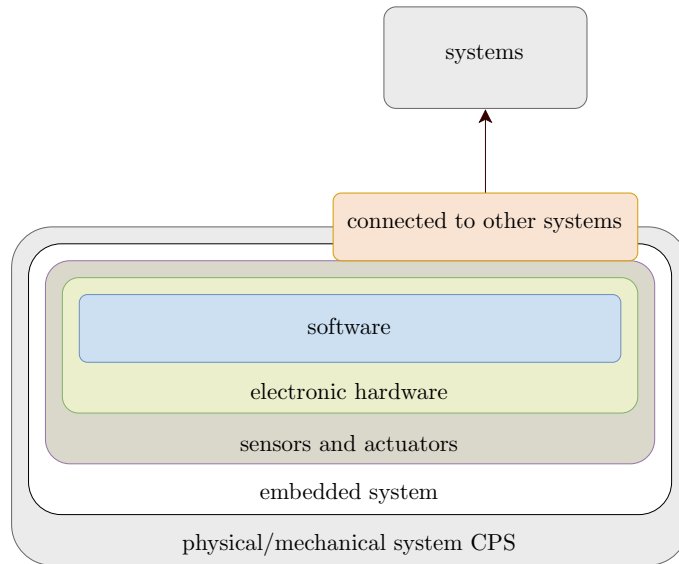


Figure 2.9: Overview of an CPS system [41]

Each system consists of several hardware or software components. In general, each **component** represents a modular, deployable, and replaceable part of the system that encapsulates implementation and exposes a set of interfaces [42]. This notation is recursive which means, that the e.g. software within a cyber-physical system can also be assembled by components. Software in combination with electronic hardware, sensors and actuators lead to a “conventional” embedded system. A cyber-physical system however extends an embedded system by the ability to communicate with other systems in its environment.

Considering the processor configuration of such a cyber-physical system, a software system (e.g. operating system) is responsible for the management of the available computation units (i.e. cores). A certification of hardware is principally feasible but not relevant for the context of this thesis. The cyber-physical system furthermore can be understood as an usual computing device (i.e. memory management unit available). For the remaining work a multi-core processor will be assumed.

The multi-core processor configuration as depicted in figure 2.10 is relevant for further constraints (e.g. criticality, task, deployment) and thus makes this decision to the most fundamental one. According to Schirmeister [125], there are two principal design

opportunities for the resulting processing model. symmetric multi-processing (SMP) and Asymmetric multi-processing (AMP). In a symmetric multi-processing system with several processors, each of them has the same architecture, uses a shared memory space with others and runs the tasks of the same operating system instance (see figure 2.11). In an asymmetric multi-processing system, however, processors may have the same or a different architecture, each processor has its own address space (i.e. distinct communication facility required), and each processor may or may not run an operating system. The AMP offers a great flexibility and supports the safety relevant aspects, but also introduces a more complex architecture where the execution of the tasks are bound to some restrictions. SMP, on the other side, is a simple model whereas a temporal isolation and partition (relevant for safety applications) through hardware is not fulfilled. Further, a combination, also known as hybrid approach, where both models are combined to introduce the benefits of both in the resulting processing design. In combination with the mixed-critical constraint, tasks are not allowed to be deployed on each core equally. A further constraint regarding the processor configuration is the affinity of task and core, where the affinity needs to be respected by other system parts like the scheduler to circumvent a break or hazardous behavior in the system and in turn enables guaranteeing a distinct or intended system behavior.

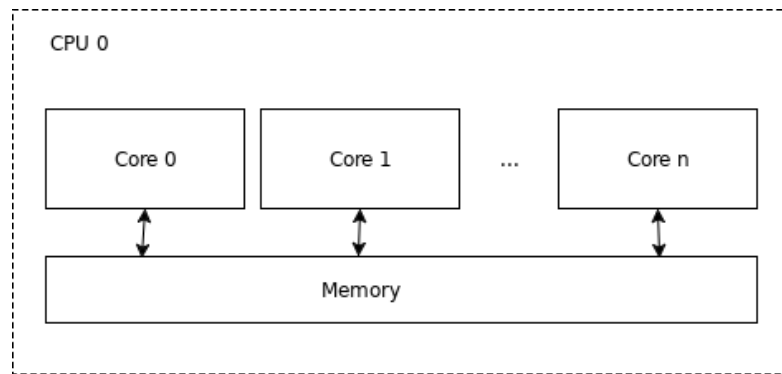


Figure 2.10: Simplified Architecture of a Multicore Processor

Commercial off-the-shelf (COTS) hardware predominantly follows the SMP model, resulting in embedded devices with a great performance and an arguable price. Of course, these systems lack the hardware support for safety related applications. Nevertheless, the intended protection/separation mechanisms can be, at least the hardware independent ones, implemented in software (e.g. Software Transactional Memory). This means a given homogeneous SMP-based hardware architecture is combined with a heterogeneous software approach, resulting in a hybrid approach (also referred as hybrid SMP-AMP). The great benefit is to allow the application of these embedded devices in mixed-critical application domains (as long as the specification provisions the protection mechanisms through software). For the combination of software and hardware in a proper way to establish a partitioned architecture, there exists several design models. Moyer [106] outlines some of the common practices for the partitioned implementation, where the

combination possibilities vary from OS-per-core, multiple SMP (coexistent OS manages multiples cores), SMP and RTOS to SMP and bare-metal system configurations.

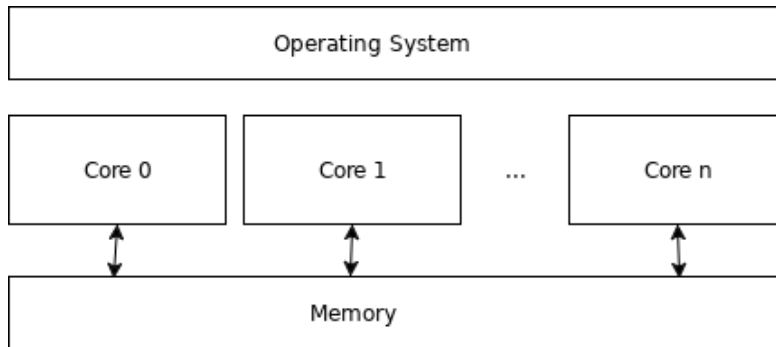


Figure 2.11: SMP Processor Configuration with Operating System

A combined design is leaned on the hybrid SMP-AMP configuration, where several operating systems are responsible for multiple cores in the (many-core) system. In contrast, the scheduler policies rather than the operating systems differ from each other. To realize this, a microkernel approach is enabled whereas the scheduling policies are heterogeneous across the cores. So, the microkernel manages all cores as usual in SMP-style but each core embodies its own scheduler. The demanded controlling (e.g. manages the partition/separation) part is outsourced as a user-space server application which can be executed on a dedicated core on the same system or decoupled as part of another system. This coexistent scheduler scheme enables the usage of several schedulers on one processor. Each scheduler manages one core and thus enables the application of algorithms and methods/strategies known from the uni-processor scheduling where no huge heuristics or complex strategies are required. Moreover, the Trusted Computing Base (e.g. SuOC) keeps small. Furthermore, the overhead of using and managing scheduler strategies in contrast to operating systems can be reduced. A critical part of the outlined approach is the admission of threads to the partitions and the guarantee of system behavior in spite of dynamic changes.

In the context of the hybrid AMP-SMP system, affinity is used to pin the thread on one specified core. This allows a fine controlled deployment (e.g. OC components are executed on one dedicated core exclusively). Due to the strict core to thread binding, a performance benefit is achieved caused by no need for switching the thread to other cores. Indeed, it needs to be ensured that the thread is executed on the same core at a given time, even when the system contains homogeneous scheduling strategies on some of its cores. Moreover, threads as part of a task set (e.g. precedence) are required to be executed on one core to ensure the execution order. The usage of an affinity allows the binding of threads to cores and thus to guarantee the execution locality. But the binding of threads to cores is restricted to a further constraint which keeps the criticality aspect of the system into account.

To sum it up, this section describes the basic resource usage of a cyber-physical system by a software system like an operating system. The basic processing management on the

operating system side is done by a scheduling mechanism. The basic terms of (real-time) scheduling will be described in the next step.

2.4.2 Real-time Scheduling

This section explains the fundamentals about the real-time scheduling used inside of an operating system for the task processing. An important aspect is the basic taxonomy for the classification of a scheduler. Followed by details about task relevant information, this section closes with a comparison of scheduling policies.

Classification of Scheduling Algorithms

In this section, the various classification aspects of a scheduling algorithm will be described. The explanation follows the diagram depicted in figure 2.12. According to the shown graph, a (real-time) scheduler can be classified according to three greater areas. At first, there are the considered timing constraints, namely soft real-time or hard real-time. Each of these categories can be divided according to the point in time when a scheduling decision takes place which is either static or dynamic. Both types of scheduling decisions can be applied in the context of a non-preemptive or preemptive scheduling mechanism.

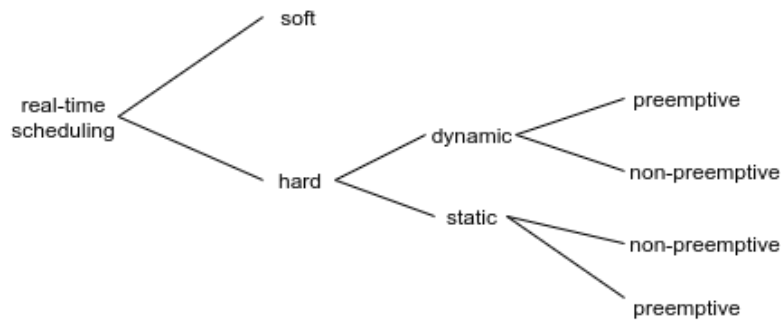


Figure 2.12: Taxonomy of real-time scheduling algorithms [88]

Before starting with the scheduling classification a definition of real-time needs to be given:

Definition 31 *A system is **real-time** if the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical time when these results are produced [88]*

Starting with the timing constraints of a scheduling mechanism, depending on to be schedulable tasks, timing demands can range from soft to hard real-time. Buttazzo [29] therefore distinguishes the following three categories of real-time where each category suffers differently from a deadline miss.

- **Hard:** A real-time system is said to be *hard* if missing a deadline may cause catastrophic consequences on the system under control.

- **Firm:** A real-time system is said to be *firm* if missing a deadline does not cause any damage to the system, but the output has no value.
- **Soft:** A real-time system is said to be *soft* if missing a deadline has still some utility for the system, although causing a performance degradation.

To explain the static and dynamic scheduling characteristics, it is important to mention that these characteristics concern the scheduling decision directly. Lee [95] describes the scheduling decision as a decision to execute a task with the following three parts:

- **assignment:** which processor should execute the task
- **ordering:** in what order each processor should execute its tasks
- **timing:** the time at which each task executes

In accordance to this, all three decisions can be made at design-time (offline) or at run-time (online). Considering the taxonomy of online and offline, Buttazzo [29] describes both terms as follows:

- A scheduling algorithm is used **off-line** if it is executed on the entire task set before tasks activation. The schedule generated in this way is stored in a table and later executed by a dispatcher.
- A scheduling algorithm is used **on-line** if scheduling decisions are taken at run-time every time a new task enters the system or when a running task terminates.

The different types of possible schedulers are derived from the point in time when the decisions are made. An off-line scheduler is often used as a synonym for static order scheduler which performs the task assignment and ordering at design-time [95]. “In general, static algorithms are those in which scheduling decisions are based on fixed parameters, assigned to tasks before their activation” [29]. On the other side an on-line scheduler, also known as fully-dynamic scheduler, performs all decisions at run-time [95]. Buttazzo [29] generalizes this taxonomy: “Dynamic algorithms are those in which scheduling decisions are based on dynamic parameters that may change during system evolution.”

The last classification characteristic of a scheduler is whether it is allowed to interrupt the currently executed task for its scheduling decision or not. The preemption can be divided into the following two cases [29]:

- In preemptive algorithms, the running task can be interrupted at any time to assign the processor to another active task, according to a predefined scheduling policy.
- In non-preemptive algorithms, a task, once started, is executed by the processor until it finishes its execution or it arrives the end of its scheduled time slice. In this case, all scheduling decisions are taken as the task terminates its execution.

Timing Characteristics of Tasks

This section covers the most important software entity treated by the task management within an operating system, namely tasks. At first, the definition of a task and its relation to synonymous terms (e.g. process) is given. Next, the relevant timing characteristics of a task will be covered.

In the context of this thesis, the term of a **task** is used as synonym to a *process* and consists of one single **thread**. A task represents a sequential computation on the CPU. Important is that the described tasks are entities which are user-space applications (i.e. managed by the corresponding management unit) and are not to be confused with kernel tasks or kernel threads (which follow a similar definition).

Concerning the regularity of a task activation, there are basically two types which can be distinguished:

- A **periodic** task is mostly triggered in regular intervals by a clock mechanism like a timer (*triggered by time*)
- An **a-periodic** or **sporadic** task occurs irregularly during system execution and is mostly triggered by an event (*event triggered*)

An overview about several timing parameters of a task execution τ_i is shown in figure 2.13. The terms mentioned in the figure are defined according to [29, 95] as:

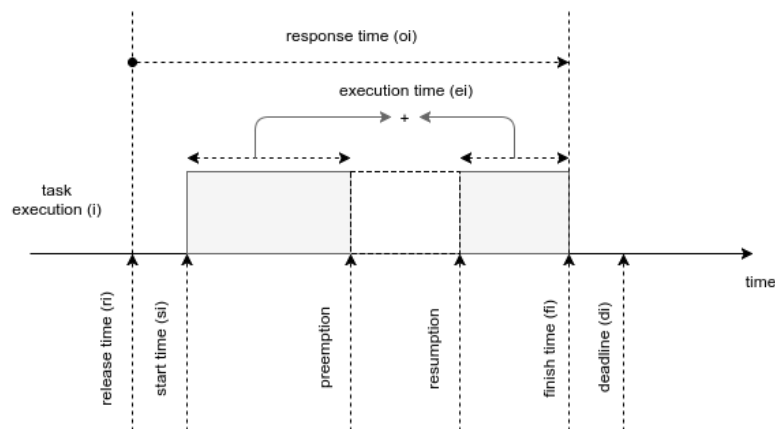


Figure 2.13: Summary of times associated with a task execution [95]

- **release time** (also arrival time) r_i is the time which a task becomes ready for execution
- **start time** s_i is the time at which a task starts its execution
- **finish time** f_i is the time at which a task finishes its execution
- **response time** o_i is the difference between the finishing time and the release time:

$$o_i = f_i - r_i$$

- **execution time** e_i is the total time that the task is actually executing on the processor
- **absolute deadline** d_i is the time by which a task must be completed
- **relative deadline** D_i is the difference between the absolute deadline and the release time: $D_i = d_i - r_i$
- **criticality** is a parameter related to the consequences of missing a deadline (see chapter 5 for more details)
- **value** v_i represents the relative importance of the task with respect to the other tasks in the system (used for optimization see chapter 5)

After outlining the scheduling basics within real-time systems, it follows the description of several aspects of finding a schedule.

Comparing Schedulers, Analysis and Tests

The comparison of schedulers includes several aspects where as a first step the general terms for the comparison of scheduling mechanisms will be defined. Followed by an important aspect of the scheduling configuration in the form of a placement. In the context of this thesis the definition of admission and enforcement is important. As part of the enforcement phase, relevant scheduling approaches/policies will be discussed in detail. The admission includes the analysis in the form of tests to produce a valid schedule where relevant details about the tests will be given.

The scheduling problem in its general form has been shown to be NP-complete. To reduce the complexity of the problem to be solvable in polynomial time, it is possible to use several assumptions (e.g. fixed priorities). These assumptions are used to classify the various scheduling algorithms. The goal is to produce a feasible schedule. In this context feasibility and schedulability can be defined as follows [29]:

- A **feasible** schedule allows all tasks to be completed
- A task set is **schedulable** if there exists at least one algorithm that can produce a feasible schedule

After describing the base taxonomy, the next aspect is the assignment of tasks to the underlying cores (i.e. mapping of scheduler on available cores). There are two approaches that can be used to realize the scheduling on multi-core systems. **Global** scheduling, where a global queue is used to manage the underlying cores by the operating system. The **partitioned** scheduling approach binds individual queues to each core [25]. There is also a combination of both possible. Hierarchical scheduling utilizes a global schedule for the overall system cores and several local schedulers on lower layers. The last approach can often be found in virtualization methods. The hierarchical scheduling approach embodies a relative overhead in comparison to both pure scheduling approaches. The

global scheduling scheme can be complex depending on the number of cores and the type of tasks which should be scheduled. Partitioned scheduling in contrast reduces the problem size and the migration overheads, while improving the predictability and the determinism reasoning of a given behavior. Also, where the hierarchical scheduling scheme can also be applied to the partitioned approach, the focus relies on a pure partitioned scheme for the system design.

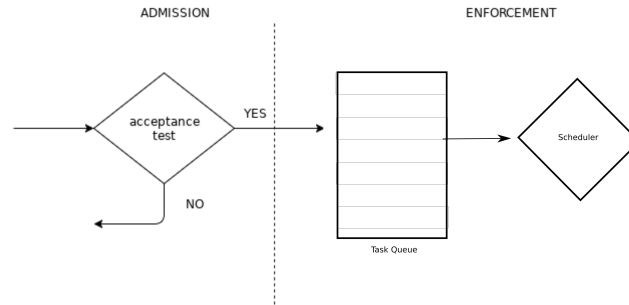


Figure 2.14: Scheme of the guarantee mechanism used in dynamic real-time systems (according to [29])

In figure 2.14 the depicted scheme represents a guarantee mechanism used in dynamic real-time systems. However, similar mechanism are applicable to static real-time systems. For the development of a consistent task management these mechanisms can be generalized. The scheduling mechanism therefore can be understood as a combination of admission and enforcement:

- During **admission**, several checks like scheduling analysis or acceptance tests are performed to verify if a request is feasible
- The **enforcement** is defined as execution of the schedule

The available tests for analyzing a feasible schedule within the admission depends on the scheduling algorithm used in the enforcement. Therefore, possible scheduling approaches will be described in more detail followed by suitable tests. The three approaches considered in this thesis correspond to the scheduling approaches described by Burns in his book [25]:

- **Fixed-Priority Scheduling (FPS)** - each task has a fixed, static, priority which is computed offline. The tasks are executed in the order determined by their priority. The priority depends on temporal requirements rather than criticality.
- **Earliest Deadline First (EDF) Scheduling** - the order of runnable tasks is determined by the absolute deadline. The task with the nearest deadline will be executed next.
- **Value-Based Scheduling (VBS)** - in situations (e.g. overload) where either priorities nor deadlines are sufficient, a more adaptive approach is required. Determining the importance of tasks in such cases is realized by the assignment of a

certain *value*. This value is used by an online scheduling algorithm to determine which task to run next.

In accordance to the presented scheduling algorithms there exists two terms which are relevant for the comparison of schedulers, namely optimal and heuristic [29].

- An algorithm is said to be **optimal** if it finds a feasible schedule, if one exists.
- A **heuristic** algorithm tends toward an optimal schedule but without guaranteeing to find it.

Both FPS and EDF are optimal with respect to feasibility. In contrast, VBS is more heuristic oriented. In the context of hard and soft real-time demands, this leads to a classification in guarantee-based algorithms and best-effort algorithms [29]. **Guarantee-based algorithms** is able to guarantee the feasibility of a schedule either in static (offline) or dynamic (online) real-time systems. This class of algorithms are able to be used in hard real-time systems. **Best-effort algorithms** however are not able to guarantee the finding of a feasible schedule which predestined them for usage in soft real-time systems.

The admission depends on the used scheduling algorithm and the intended usage either in high or soft real-time systems. In general, there exists several test methods where the most important characteristics of these tests are **sufficiency** and **necessity** [25].

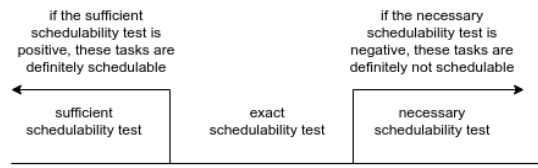


Figure 2.15: Kopetz 10.2: Necessary and sufficient schedulability test

- A schedulability test is defined as **sufficient** if a positive outcome guarantees that all deadlines are always met
- A failure of a **necessary** test leads to a deadline miss at some point during system execution.
- An **exact** test is sufficient and necessary

Figure 2.16 summarizes the terms defined in this section. A real-time system can be classified as either hard or soft. Hard real-time systems are either static or dynamic. Static systems prefer an offline admission (e.g. Response Time Analysis) and an optimal (static) enforcement (e.g. FPS). This combination leads to a highly guarantee capable system. In dynamic hard real-time systems however a priority based online admission (e.g. acceptance test) is mandatory to restrict the optimal but dynamic enforcement (e.g. EDF). In dynamic soft real-time systems, admission as well as enforcement are

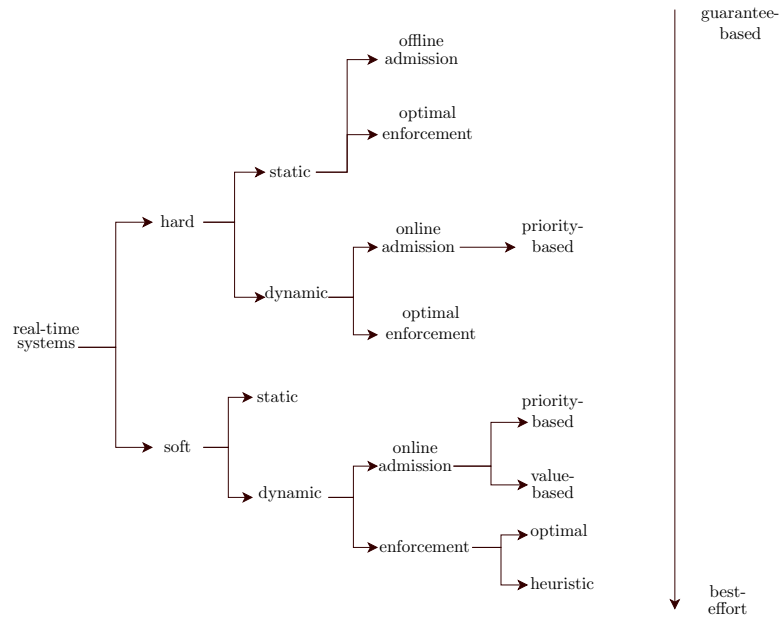


Figure 2.16: Scheduling comparison

extended by value-based or heuristic approaches respectively. In its most extreme form, a dynamic soft real-time application follows a best-effort strategy.

In cases of dynamic enforcement, the main objective is the management of overload situations. Especially in combination with EDF, an admission control procedure helps to avoid transient overload situations. For the admission procedure, the relative importance of a task is represented by a value which can be classified as follows [25]:

- **static** - a task's value stays the same whenever it is released
- **dynamic** - a task's value is recalculated at the same time when it is released
- **adaptive** - a task's value will be changed during its execution

The values are not only used for the admission but also for the optimization of a given system.

To sum it up, this section has considered the timing aspects of a cyber-physical system. At first the timing resource, core, was investigated and its principal management by an operating system. The scheduling as central operating system service for core management was explained. Several details about classification, task model and comparison of real-time scheduler followed. Some system properties however are not necessarily pinned down to timing demands. System properties related but not dependent on timing properties will be covered in the next section.

2.4.3 Communication

This section completes the system analysis with relevant communication fundamentals. Like in other domains, there are numerous communication technologies for transferring information between embedded systems. These technologies are a combination of both hardware and software solutions. There are common aspects however which clarifies the communication fundamentals without going too deep into the detail of a single solution. A simple example to demonstrate the heterogeneous communication technologies altogether can be seen by utilizing the former overview of a CPS (see figure 2.9). Possible communication could take place between the single systems of the CPS like suggested in the following communication schema:

1. communication between cores on the electronic hardware via a bus or cache
2. communication between sensors/actuators and the electronic hardware via a two-wire interface (e.g. spi) forming an embedded system
3. communication between a cyber-physical system and another system via an Ethernet-based communication bus realizing the connection to other systems
4. communication between a car (i.e. set of cps) and another car via a wireless communication (e.g. car2car)

This mixture of communication technologies is a concern in current automotive systems. The development of future cars (i.e. example for cps) however tries to establish a homogeneous communication infrastructure rather than domain-specific isolated solutions. In the long-term, Ethernet could be used to establish a homogeneous communication infrastructure, at least within a car [138]. Nevertheless, there are still situations where a communication in a heterogeneous context needs to be established. These situations however can be addressed by a standardized communication model like the Open Systems Interconnection model.

A concrete example for the communication between cars can be found in the domain of Car to Car and Car to Infrastructure communication. These systems are used for traffic control and allow the avoidance of accidents through a communication between cars. The car2car consortium, comprised by car manufacturers and suppliers, coordinates the development in this area. The consortium proposes a three-layer protocol stack as depicted in figure 2.17. As it can be seen, the consortium prefers the usage of already existent protocols and technologies rather than developing their own solutions.

In conclusion, the communication within an automotive embedded system is moving to homogeneous solutions where applicable. In other scenarios, standardized models are used to guarantee a homogeneous inter-system communication.

Conclusion

This chapter has introduced the foundations of embedded systems. The structure of the chapter hereby was oriented according to the development process of embedded systems.

2 Fundamentals of Embedded Systems Development

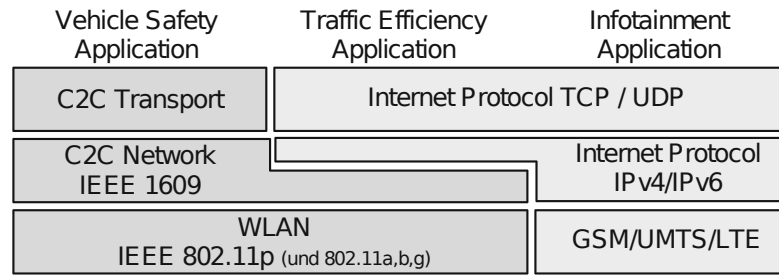


Figure 2.17: Car2Car protocol stack (cited after [138])

Where possible, automotive examples were chosen to describe the several aspects during the process. The several relevant definitions were placed at the distinct positions within the process. With this chapter, the taxonomy for the remaining thesis was clarified. As a next step, chapter 3 will provide an overview about the domain of flexible software management in mixed-critical systems.

3 Domain Analysis

This chapter will cover the domain analysis for this thesis. In a first step, available operating system architectures for multi-core base systems will be investigated in section 3.1. The following section 3.2 about real-time scheduling will investigate in several design considerations for realizing a scheduling approach capable of managing a multi-core system. This section will be complemented by the subsequent section 3.3 about mixed-critical system properties. The section 3.4 about self-adaptive software engineering will analyze the several possibilities to design and develop a self-adaptive system. Finally, the last section 3.5 in this chapter will outline several restrictions and assumptions resulting from the combination of the former sections to realize a flexible task management approach.

3.1 Operating System Architectures for Multi-Core Systems

This section describes available operating system architectures for multi-core systems. Relevant architectures suitable as a base for realizing a flexible task management are in the spotlight. Beginning with systems known from an automotive domain, this section steps through the several architecture types ranging from microkernel over monolithic kernel to middleware based management solutions.

In the automotive domain, the open standard OSEK/VDX specifies in several parts amongst others the operating system architecture. The AUTOSAR consortium reuses this specification to realize its standard. Besides the operating system, AUTOSAR also defines a methodology for the development of automotive systems. For the methodology, a toolchain support is mandatory due to the fact that a great amount of the development process is fully automated. A missing toolchain leaves the developer with the kernel source code in an inapplicable state. For the toolchain there are great variation in licenses and costs depending on selected products and vendors. With only the OSEK OS in mind, there exists alternatives to commercial products in form of open source implementations (partial with tooling support) like ERIKA Enterprise [55] or Trampoline [16]. The intended target platform of these open implementations, however, is mostly suited for embedded systems where e.g. a MMU is non existent. Where applicable, AUTOSAR OS is intended to be used in a multi-core setting [110]. Further, there exists extensions like Adaptive AUTOSAR [39] which provides a decent possibility to modify the static systems. By the usage of these extensions, however, there are still constraints like the run-time modification of schedules which makes it hard to realize a dynamic flexible task management.

Another operating system architecture commonly found in the area of real-time embedded systems is based on the microkernel paradigm. A special focus of these systems relies on the usage in scenarios with high safety or security requirements. To achieve

the demanded assurance, microkernel-based systems are designed for supporting strong separation of software components through system partitioning (also known as separation kernel). Especially, mixed-critical systems can benefit from the separation [96]. Further improvements even show the advance of a multikernel paradigm especially for multi-core systems [15] where the cores are managed by distinct kernel instances rather than one monolithic kernel. Because of this separation, the overall performance of a microkernel-based system highly depends on the throughput and therefore on an efficient implementation of the selected communication mechanism between software components (i.e. IPC) [97, 98]. This constraint as well as several other design improvements were researched intensively [67, 54] leading to improved microkernel architectures and implementations. As well known, the L4 microkernel family contains several implementations for different use cases. Some of the commercial implementations like PikeOS¹ are successfully applied in several industrial domains where other open source implementations like Fiasco.OC² or seL4³ are freely available. Except for the common architecture design, the several implementations vary strongly depending on the individual use-case which makes a direct comparison difficult [56]. PikeOS is a static configured separation kernel which fulfills the certification criterion according to several safety standards. Fiasco.OC, however, focus lies on a dynamic and adaptive system without a certification. Current research in this area lead to a new open source kernel, seL4, which provides an unique aspect of formal verification of the complete kernel as required for the usage in certified mixed-criticality systems [20, 99, 83].

Besides OSEK/VDX-based systems and microkernel-based systems, there is an ongoing approach to apply the Linux kernel in safety-critical systems [101, 77]. Focusing the required aspects of certification and real-time capabilities, the Open Source Automation Development Lab (OSADL) mainly coordinates these efforts with its main goal to bring the Linux kernel into industrial applications. Several preliminary assessments (complexity, usage and reliability) considering a potential certification were identified [117]. Considering the real-time capabilities and reliability of the Linux kernel, several extensions (commonly known as real-time preemption patch) are applied which enhances the kernel with real-time functionality [94]. Again, there are several products and implementations available which support among others great tool-support for building a custom Linux distribution. The required certification for the usage in safety-critical systems, however, is still missing.

For using a (real-time) operating system, the kernel alone is not enough. There are other software components required to provide the functionality of the operating system. Especially in the area of microkernel-based systems, there exists several approaches to provide the required components within a run-time environment. Known component architectures are CAMkES⁴ (Component Architecture for microkernel-based Embedded Systems) from the sel4 project L4Re⁵ and its predecessor L4Env from the Fiasco.OC

¹<https://www.sysgo.com/products/pikeos-hypervisor/>

²<https://os.inf.tu-dresden.de/fiasco/>

³<https://docs.sel4.systems/>

⁴<https://docs.sel4.systems/CAMkES/>

⁵<https://l4re.org/>

3.1 Operating System Architectures for Multi-Core Systems

microkernel. There also exists further architectures which enhance the idea of a run-time by providing the same components to a variety of microkernels. The Genode OS Framework⁶ for instance is a developer tool-kit which aims to provide building blocks to create customized operating systems where it supports both microkernels and monolithic ones like Linux. In combination with Fiasco.OC, the resulting system is likely to be much more secure and efficient in resource usage which allows usage in small embedded systems. Moreover, there are run-time environments which are independent from the underlying kernel and provide their services to arbitrary systems. Workgroups of the Multi Core Association⁷ develop industry standards with focus on communication (MCAPI), resources (MRAPI) and task management (MTAPI). These standards are used in the development of embedded systems to focus on the considered solution independently from the underlying hardware or software stack. For the MTAPI, there exists an open source implementation called EMB²⁸ by Siemens. Another solution is the Robot Operating System⁹ (ROS) which is based on the Unix like operating systems (e.g. Linux). An usage in the embedded domain seems principally feasible, but is restricted due to the lacking real-time support of this combination [129].

The so far considered system architectures are running directly on hardware following the classical operating system approach for managing the available hardware resources. But there is another architecture approach for future embedded systems, namely virtualization. A hypervisor resides between the hardware and its guests (i.e. virtual machines) providing the demanded hardware access by a distinct calling interface. Virtualization is a method gaining popularity in the automotive domain [109] and other embedded system domains. There exists hypervisor approaches for embedded multi-core systems which allows the application of full virtualization and paravirtualization in one common system [59]. Like any other architecture covered in this section so far, there exists different implementations and approaches for hypervisors. An interesting aspect, however, is the proposal of Heiser [69] where the application of virtualization in embedded systems can only be beneficial if an overall operating system technology shift to microkernels happens. An implementation of such support within a microkernel is presented by executing Linux and Android on top of a Fiasco.OC microkernel¹⁰. Occurring limits introduced by the hierarchical scheduling model are attempted to be circumvented [93] whereas modifications within the guest are required which lead only to partial improvements considering the application in embedded systems.

This section has described the several operating system architectures which can be found in embedded multicore systems. For the realization of a flexible task management there exists several operating system choices where each approach offers its own strength. There are systems which differ mainly from the used kernel architecture, run-time environment or virtualization. Scheduling, however, can be seen as the least

⁶<https://genode.org/>

⁷<https://www.multicore-association.org/>

⁸<https://emb.io/>

⁹<http://www.ros.org/>

¹⁰<http://l4android.org>, <http://l4linux.org>

common denominator of all approaches. The next section, therefore, steps further in this direction and presents recent research results in the area of real-time scheduling.

3.2 Real-Time Scheduling of Dynamic Mixed-Critical Systems

This section covers the variety of real-time scheduling approaches found in the domain of dynamic embedded mixed-critical multi-core systems. Starting with suitable scheduling architectures, several approaches in particular hierarchical and microkernel-based scheduling will be investigated. This will be followed by scheduling mechanisms suitable for dynamic systems. After that, scheduling for multi-core systems is considered. This section closes with possible scheduling policies capable for the usage in mixed-critical embedded systems.

In the domain of dynamic systems, there are situations where neither priority nor deadlines are sufficient for a scheduling decision. With the advance of dynamic systems there was an early demand to develop new approaches which are able to handle dynamic workloads. One of the early first steps in this direction was the combination of known scheduling algorithms. Stewart [128] describes in his work the Maximum-Urgency-First algorithm (MUF) as a combination of fixed and dynamic priority scheduling. With an urgency value per task he was capable of showing the superiority of the combination in contrast to traditional approaches like rate monotonic scheduling. Several deficiencies of the original MUF design considering deadline misses (e.g. overload situations) are addressed and optimized in consecutive works [135, 122]. In his paper, Burns [28] highlights the importance of a value-based scheduling approach in the context of flexible scheduling and adaptive systems. So, independently of further system characteristics like criticality, scheduling of a dynamic system requires the introduction of a greater, fine-grained value-based decision mechanism allowing adequate system modifications. Due to the fact that many scheduling policies are available, the question arises: which combination is sufficient for the scheduling of adaptive systems?

Especially in real-time systems, rate-monotonic schedulers are often used due to their straightforward implementation and supposed benefits compared to more dynamic schedulers like EDF. But the drawbacks of EDF versus RM are not as huge as guessed. Indeed, Buttazzo [30] states in his work that many assumptions about the superiority of RM are either wrong or not valid in all usage scenarios. He concludes that only in hard real-time scenarios RM can provide a small benefit against EDF where in every other scenario EDF is equal or better than RM. Considering the implementation issues of EDF, Buttazzo further demonstrates a possible usage in embedded systems by an efficient EDF implementation [31]. Proposing EDF and FP as the two main scheduling policies with the domain of real-time systems, Burns [27] combines both approaches to create a hybrid flexible scheduling policy which benefits from EDF's efficiency and FP's predictability. Nevertheless, there are situations (i.e. overload) where the decision cannot be based on the priority alone. Further in his work, Ravindran [118] states that the usage of deadline/priority-based specification of time is inadequate. He therefore argues that time/utility functions and the corresponding scheduling paradigm provide a more

generalized, adaptive and flexible approach. A potential scheduling policy is presented by Kluge with the value-based MKU scheduling algorithms [85]. This heuristic algorithm allows a proper handling of tasks even in overload situations whereby a defined number of tasks need to be executed to guarantee the overall system behavior. After investigating the possible scheduling combinations, it is interesting how these combinations can be designed.

Considering the usage of a resource reservation approach, many concepts were proposed that are following the idea of a container-based server design [29] which subdivides the CPU in time slots (i.e. container with capacities). A runnable task is assigned to one of these slots. A consecutive approach is the deployment of scheduling policies on several layers. On the global layer, one scheduling policy (e.g. periodic server) schedules the tasks according to a container capacity whereby each task represents another scheduler (i.e. local layer). A hierarchy of schedulers arises where this scheme is commonly known as hierarchical scheduling. The hierarchical scheduling concept does not make any considerations about the concrete scheduling policies on the global or on the local layer. An approach of combining fixed-priority scheduling policies on both layers is presented in the work of Davis [43]. Other possible combinations of local EDF and either FP or EDF as global scheduler are also investigated in the work of Zhang [136]. There are also hierarchical scheduling approaches that foresee more than two levels which can also be adapted via a controller based mechanism [81, 80]. Hierarchical scheduling thus allows a software-based separation of tasks within one system by simultaneously achieving a flexible scheduling approach. An interesting point left is how different operating systems are using the concept of a hierarchical scheduling mechanism.

Integrating hierarchical scheduling in different system architectures enables these systems a concurrent execution of their tasks either in a single-core or a multi-core setting. Asberg [7] describes in his work mandatory steps to integrate a hierarchical scheduling into the AUTOSAR OS. His work primarily addresses the challenge of component isolation at run-time. He therefore suggests the usage of a Hierarchical Scheduling Framework (HSF) and foresees the extension of AUTOSAR's component model by the notion of subsystems/partitioning. With a missing partitioning concept in AUTOSAR, it is hard to achieve the approach proposed within this thesis. Within the domain of virtualization, hierarchical scheduling also plays an important role. Groesbrink [63] demonstrates in his work the usage of a partitioned hierarchical scheduling approach where a partitioned RM scheduling strategy on the global level is used. The hosted OS within each VM schedules its tasks according to their own local scheduling policy. This approach is especially interesting for multi-core settings. Using the hypervisor based approach, however, the VMs are treated as black boxes where a modification of hosted OSs is required to provide the underlying global scheduler with the required timing information. A similar approach for the Linux kernel is investigated by Asberg [6]. In the context of microkernel-based systems, Lackorzynski [93] realizes this approach for the L4 Fiasco.OC microkernel. Similar concepts of hierarchical scheduling were also investigated in the domain of microkernel-based systems but with another attempt. Replacing the hypervisor with a microkernel, there are several efforts to move scheduling decisions from kernel-space to user-space [130, 5] for improving the flexibility of a microkernel. Hierar-

chical scheduling allows the concurrent execution of tasks which can be supported by the underlying hardware platform through the usage of multi-core processor configurations.

There is a trend in the automotive industry where ECUs become more and more multi-core systems and new challenges for finding adequate scheduling concepts arise [104]. A first work on utilizing a co-processor for scheduling in real-time systems is proposed by Hildebrand [72] which allows the execution of time consuming computations apart from the remaining system. A potential overhead is reduced by this approach. A survey of hard real-time scheduling for multiprocessor systems is given by Davis [44]. In his work, he compares the several scheduling approaches for multi-core systems and identifies the current state but also opens issues for further research. Especially, the benefits of partitioned and global scheduling against each other are outlined. Both scheduling approaches have clear benefits and it depends on the use case which one to prefer. In summary, however, it is proposed that the partitioned scheduling approach can outperform the global approach even in hard real-time scenarios. In his work, Baruah [14] considers the scheduling of mixed-criticality implicit-deadline sporadic task systems on identical multiprocessor platforms. He compares the two approaches of global and partitioned scheduling where the partitioned approach outperforms the global approach in his simulation. The issues addressed, among others, are the overheads as an important aspect in multi-core scheduling. The usage of a partitioned scheduling approach is also proven in the context of mixed-criticality task sets by the work of Kelly [78] where the task allocation and the priority assignment are focused on a fixed-priority scheduling approach. For this work, several combinations of allocation and assignment algorithms are considered with the result that an ordering according to a task criticality in combination with an efficient priority assignment algorithm outperforms the other combinations. Heterogeneous multi-core systems are also subject to the research where each core is dedicated to a separate scheduler with different scheduling policies [34] extending the former idea of a co-processor. However, scheduling policies alone can only provide the core of a flexible task management. There are further mechanisms required.

The so far described combination of scheduling policies is missing an important aspect. Basically, dynamic mechanisms are allowed to adapt the existing task sets (e.g. according to their capacity demands). By choosing a partitioned scheduling paradigm, a dynamic assignment of tasks to corresponding cores will be resolved during run-time. Both aspects are not covered by the presented combination of flexible scheduling approaches. For supporting these aspects with the given scheduling approaches, an online schedulability test needs to be added during run-time. The continuous adaptation of task sets further requires the usage of a feedback control mechanism which monitors the system and delivers required information for further tests. Each test method needs to be efficient because it is generally executed during run-time. In his work, Santos [123] investigates efficient on-line schedulability tests for feedback scheduling of soft real-time tasks under fixed-priority. He concludes that the choice of a proper method for testing depends on the kind of system and the hardware performance.

Further investigations in the context of schedulability analysis with focus on EDF are presented in the work of Zhang [137]. His work results in the Quick convergence Processor Demand Analysis (QPA) algorithm for analyzing the feasibility of a schedule

in single-core systems. A further comparison of existing schedulability tests for global EDF scheduling on a multi-processor platform is presented by Bertogna [18]. He proposes an algorithm that combines the advantages of the existing techniques. For fixed priority scheduling policies, Negran [111] considers a response time analysis for a partitioned multi-processor real-time system. The aim of this work is to soften the so far taken constraints of previous work (e.g. usage dependent tasks). Especially in the context of partitioned scheduling, admitting a new task plays an important role. Masrur [100] presents an algorithm for the admission control for a partitioned EDF scheduler which can be executed in constant-time i.e. independently from the input size of the to be analyzed task set. In his work, Becker [17] investigates the usage of not only a single test rather than a combination of sufficient and exact test. He demonstrates this new combination in a simulation-based environment. The so far investigated schedulability tests and schedulability policies have in common that their analysis is based on timing characteristics.

The introduced task sets impose distinct timing demands on the system. Dependability requirements however are orthogonal to timing demands (e.g. high critical tasks are mostly hard real-time) but not similar. Criticality can therefore not exclusively be expressed by time. Moreover, tasks with different criticality demands lead to the challenge of scheduling a mixed-criticality system. Thus, the question arises, which additional criticality scheduling mechanisms are available? In his paper, Brandt [22] investigates in the combination of hard real-time, soft real-time, and non-real-time processes with the intention to provide demanded flexibility and guarantees for such systems. For developing his scheduler approach, he follows the Resource Allocation/Dispatching (RAD) model which separates the resource management from the timing of delivery of those resources. As a proof-of-concept, Brandt has developed the Rate-Based Earliest Deadline (EDF) scheduler which provides the integration of processes with several timing demands (e.g. hard real-time). Under the usage of a novel bandwidth reservation mechanism (i.e. constant bandwidth server - CBS) with an EDF scheduling algorithm, Abeni [2] provides a solution for dynamic systems where several tasks with different criticality can be executed side by side (i.e. temporal isolation) on one system under the guarantee that a predefined bandwidth will never be exceeded. Baruah [13] provides a formal model of mixed-criticality workloads combined with analysis methods and corresponding algorithms. The scheduling algorithms he investigates are reservation-based scheduling and priority-based scheduling. He concludes that priority-based scheduling is superior to reservation-based scheduling measured by the processor speedup factor. A side-by-side comparison of mixed-criticality scheduling is presented by Huang [73]. He considers the comparison between priority assignment, period transformation, and zero-slack scheduling. A further article of Huang [74] extends this comparison by considering the several available user-space enforcement and priority assignment mechanisms within the domain of mixed-criticality scheduling. He concludes that those new approaches can provide substantial improvements in task schedulability over the mixed-criticality scheduling approaches which were investigated previously. As a key observation Huang proposes that their developed version of an adaptive mixed-criticality enforcement mechanism in combination with a fixed priority scheduling outperforms the other tested combina-

tions within the chosen scenarios. All tests are performed on a single-core based Linux platform. In his work, Völz [133] investigates in mixed-criticality algorithm support for fixed-priority scheduling within the context of a microkernel. He proposes that all investigated algorithms can be mapped to the microkernel-internal scheduling structure and thus demonstrates the feasibility of supporting mixed-criticality workloads within a microkernel. Up to now, the assumption is that criticality workloads cannot be managed by a real-time scheduler. However, there are efforts to enable such a management.

The following papers are discussing the requirements and demands of several extensions to conventional real-time scheduling approaches considering the new resource allocation and scheduling challenges which comes with the advance of mixed-critical systems. In their work, Baruah and Vestal [11] present their finding about certain schedulability aspects considering a novel task model for multi-criticality real-time tasks implemented upon preemptive uni-processor platforms. They derive and evaluate a new hybrid scheduling scheme to circumvent the limitations of current scheduling schemes in this area (i.e. EDF). Guo [65] targets the problem of representing the failure probability of a safety critical real-time system which is required for the certification process within a mixed-critical task model. He therefore extends the mixed-criticality task model by adding a new parameter to each task which represents the likelihood of all jobs of a task finishing their executions. A novel EDF-based scheduling algorithm exploits these probabilistic information to make its decisions. Baruah [12] identifies challenges of mixed-criticality systems considering both the static schedulability analysis and run-time monitoring. He proposes a new implementation scheme for fixed priority uni-processor scheduling of mixed-criticality systems with the requirement that jobs monitor their execution times. For their evaluation, they study two scheduling algorithms: one that assumes limited run-time support and one that requires additional run-time support for mixed criticalities. They conclude that an additional run-time support provides superior schedulability/certifiability guarantees.

This section has given a brief overview about the considerations in the area of real-time scheduling. It therefore identifies the different aspects of scheduling mechanisms and their influential factors from several system requirements. Starting with “classical” scheduling problems the section evolves to topics considering the scheduling in multi-core configurations and mixed-criticality systems. However, there are distinct aspects of mixed-criticality systems that are not directly addressable by scheduling mechanisms. The following section will cover these aspects in more detail.

3.3 Mixed-Critical System Properties

Considering the (fair) resource management between high-critical and low-critical tasks, this section will summarize the related research to this topic. Beginning with the fundamental research challenges within this area, the main aspect of resource management will be outlined. This is followed by common approaches which allow a fair usage of resources between components with different criticality levels. After that, system issues

like architecture design will be covered. The section closes with the research regarding an operating system support.

The work of Burns [26] presents an actual state of the art report in this research area. Burns identifies the fundamental research question in this area as a trade-off between the partitioning (safety) and the sharing for efficient resource usage. Further, he identifies several system problems relating to design and implementation of hardware and software run-time controls.

Almeida [3] describes in his work the adaptation of a safety-critical system within well-defined spaces. He follows an open systems approach but delimiting the space of possible adaptations. As a central element he defines configuration spaces which constrain the possible adaptation and thus ensure safety. For changing from one configuration space to another, Almeida utilizes a switching logic during run-time.

A better resource management without a degradation of low-critical tasks is important. Where low-criticality tasks should not be mistaken as tasks with little value and simply dropping of them when a criticality change occurs is unacceptable [58]. A method to allow tasks with lower criticality to be executed as long as they do not prevent higher criticality tasks from meeting their deadlines is proposed by Santy [124]. For demonstrating his concept he uses a fixed priority scheduling strategy. Santy's method is denoted as Latest Completion Time (LCT) where experiments show a reduced number of tasks that need to be suspended. Rather than reducing the overall suspending of lower critical tasks, Jan [76] uses the elastic task model to maximize the execution rate of low criticality tasks. He uses the slack time generated by over-provisioned high-criticality and the low-criticality tasks to maximize the execution. Additionally, Fleming [58] proposes a scheme to maintain the operation of lower criticality tasks as long as possible. For that reason he "introduces the notion of importance as a means to provide a greater level of control and guarantees for LO criticality tasks during a criticality change".

Groesbrink [62] proposes an architecture supporting an adaptive resource assignment to virtualized mixed-criticality systems. His approach deals with the heavily under-utilized resources caused by pessimistic assumptions and static resource management approaches. The developed resource management protocol for virtualization respects criticality levels and allows the addition of subsystems at run-time. The flexible resource management approach provides a solution for open virtualized mixed-criticality systems. It allows a better utilization of resources through the dynamic adaptation of resource assignment triggered by either application behavior or environmental changes. As a prototyping platform, a custom real-time hypervisor, Proteus, is used. In further research, Groesbrink [61] investigates in partitioning of virtual machines onto a multi-core platform. The partitioning problem therefore is expressed as a formal model and an algorithmic approach which systematically leads to candidate solutions. The overall optimization goals are minimizing the required number of cores or maximizing the distribution of critical virtual machines. According to Groesbrink, the proposed automated "solution guarantees to find an optimal candidate, scales well with regard to an increasing number of both virtual machines and processor cores, and provides analytic correctness, which can support system certification".

The work of Anderson [4], as one of the first in this area, “describes the development of operating-system support for enabling mixed-criticality workloads to be supported on multi-core platforms”. In this work he proposes an architecture for the support of periodic task systems in the context of LITMUS^{RT}. Especially, he addresses the problems regarding the interference between high-critical and low-critical tasks in the context of scheduling. He identifies two fundamental types of algorithms in the scheduling domain, scheduling decision and test schedulability. Moreover, he is categorizing the partitioned scheduling approach as hard real-time capable and the global scheduling approach as soft real-time. Further, the interference between high-critical and low-critical tasks targets the need for temporal isolation which can be assured by the operating system by providing a container abstraction. Anderson transfers this container concept to a server-based approach with resource budgets that is based on an idea which is formerly presented by Abeni [1] for uni-processors.

The work of Mollison [103] supplements the work of Anderson by extending the formerly described server-based approach with slack stealing. Additionally, Mollison outlines a hierarchical architecture where each criticality level gets its own dedicated scheduling strategy assigned (i.e. criticality level is directly mapped to scheduling strategy). So, for level A (highest criticality), a table-driven or cyclic scheduling is used where level B is scheduled by partitioned EDF, level C and D both are scheduled by G-EDF (global EDF) and level E by Best Effort.

Especially for the introduced overhead, the work of Herman [71] corresponds to the work of Anderson and Mollison. The work presents the overhead for the release of an task and the actual scheduling overhead. Herman postulates that the usage of LITMUS^{RT} for testing various RT-related design alternatives makes it attractive, but being based on Linux disables it as a viable candidate for actual deployment in such systems.

To sum it up, this section has covered the ongoing research considerations regarding the vital aspect of resource management in mixed-critical systems. The several aspects like criticality levels, notion of importance, and system integration for realizing a resource management mechanism were addressed. A point left for the next section is the invention of flexibility within resource management. The domain of self-adaptive systems therefore will be investigated in further detail.

3.4 Self-Adaptive Systems Properties

This section will consider the flexible aspect of resource management in the context of self-adaptive systems. The section starts with common research challenges within this area and identifies the design and documentation of self-adaptive systems as a key challenge where several approaches for designing systems with self-x properties will be outlined. This is followed by a short summary of related work about applying self-adaptive systems in safety critical domains. The section closes with concrete applications of self-adaptive approaches in operating systems.

It is a challenge to document a self-adaptive mixed-critical architecture. Self-x architectures in common lead to highly dynamic context-aware systems as a reaction on

context changes. The overall system manages its components dynamically (remove, add, modify). The components realize a different behavior according to its context. For the development of self-adaptive systems, architectures play an important role. There are several architecture patterns for the description of self-adaptive systems available. In his work, Cakar [32] identifies the lack of a clear definition of self-organization considering the design of self-organizing technical systems. He therefore investigates a survey of currently identified research challenges and presents an approach to a quantitative definition of self-organization that is applicable to technical systems. Many aspects regarding the software engineering for self-adaptive systems are collected by Cheng [35]. Starting with a research road map, Cheng identifies the common open questions according to self-adaptive systems. Besides the general challenges for finding a common taxonomy for systems with self-x properties, there are several efforts for finding an adequate model-based description for such systems.

A reference model for self-management can be found in the work of Kramer [89]. Additionally, research challenges in this area are covered. Basically this model is based on a three layer architecture model which, beginning with the lowest layer, consists of component control, change management and goal management. For the component control layer, the main task is to add or remove components to or from the system, which is identified as a challenging task because of uncertainty during changes. A safe application preservation is demanded through the implementation, thus a change does not lead to a undesirable transient behavior. Change management, as the middle layer, executes several actions as a response to a changing environment. The most challenging task on this level is dealing with distribution and decentralization. The highest layer, goal management, consists of time consuming computations such as planning. Especially in the area of embedded systems, efficient planning algorithms need to be found. So, for the description of a self-managed system, a component based approach embedded in a three layer architecture is a possible description metric for such systems.

Another interesting work for designing architectures supporting self-x properties can be found in the work of Oreizy [114]. Again, components can be added, removed or replaced in the system. He introduces the concept of an observer and a dedicated synchronization component. The latter is required, because of the observation that adding a component to a running system, it must not assume that the system is in its initial state. Additionally, his work describes the usage of components and connectors. The latter ones bind components together and build up a well defined interface. Based on his research, Oreizy [115] describes the tasks of a so called adaptation management (i.e. controlling the adaptation) in more detail. "It monitors and evaluates the application and its operating environment, plans adaptations and deploys change descriptions to the running application" [115]. Moreover, in the same work Oreizy describes the need for the management of goal changes (so called evolution management). The concept of an adaptation management has influenced other works in the area of autonomic computing that are based on the concept of a cycle for monitoring, analyzing, planning and executing (MAPE) of tasks [38].

Cheng [37] introduces the concept of styles for describing different architectural aspects of the system. He describes the usage of central elements like components, connectors

3 Domain Analysis

and their interfaces (ports and roles). Moreover, Cheng defines so called representations to support various levels of abstraction and encapsulations. Now, the logical grouping of components according to a dedicated system configuration is possible. In further work, Cheng [36] describes the requirements for the design of self-adaptive systems to be an engineering solution. He proposes that the adaptation logic can be extracted from the actual system code and treated as separate from the system.

After covering the capabilities of designing systems with self-x properties, there are architecture patterns which can be used as a starting point. Richter [120, 23] proposes a generic observer/controller architecture for Organic Computing. With his architecture-oriented design approach, he wants to allow the self-organization by simultaneously controlling the emerging global behavior of this kind of systems. He defines three distinct roles within his architecture: a system under observation and control, observer and controller. Every role is defined with details about their sub-components and their expected functions. The outlined architecture is meant to be a framework for different algorithms stemming out of other domains like machine learning. Current challenges in the domain of Organic Computing are addressed in the work of Schmeck, Müller-Schlör, and Tomforde [127, 107, 132]. Primarily, a common understanding of self-organization is a central problem. Moreover, the design and control mechanism of self-organized systems embodies a great research challenge. A similar but earlier approach to Organic Computing can be found in the domain of Autonomic Computing, where data centers are the main domain for this approach. Kephart [79] envisions the concept of autonomic computing with the MAPE-cycle as its core. A concrete realization of this architecture for instance is presented by Kluge [87, 84]. In his work, he proposes a real-time operating system architecture with inherent support for Autonomic and Organic Computing. Based upon this real-time operating system, Kluge introduces an Autonomic Management mechanism which allows the separation of real-time applications from none real-time reactions.

Enabling the usage of self-adaptive architectures in a safety-critical environment concerns the trade-off between flexibility and assurance where the following related work proposes possible solutions for this problem. Considering a possible management architecture for building adaptive real-time systems, Kluge [86] extends the observer/controller architecture for an application in embedded real-time systems. He proposes a two-layered management architecture where the global management is similar to previous concepts in autonomous computing (i.e. MAPE). On the lower level, small management units are used for micromanagement operating on a small parameter set. The basic idea is to react on the lower level as fast as possible and defer complex and more sophisticated reactions on the global management layer. Fischer [57] addresses the problem of ensuring correct self-reconfiguration in safety-critical applications. He proposes an approach for verified result checking to apply only valid results to the system. Especially in safety-critical applications, the uncertainty introduced by self-x algorithms can be addressed. His approach verifies the results at design time independently from the algorithm used at run-time. Further, he integrates the approach in a organic computing architecture. As a result, the architecture allows the usage of self-x algorithms, but is still being able to proof correctness of its results. A brighter overview about assurance

3.5 Differences between Related Work and Proposed Approach

in self-adaptive systems is given in the work of De Lemos [45] where he identifies several key research challenges. The three major topics he focuses on are perceptual assurances, composing and decomposing assurances and control theory assurances. Identified research challenges consider the quantification of uncertainty, methods for also adapting the associated assurance cases and the definition of clear guidelines enabling the application of control theory principles into self-adaptive systems.

This section has outlined the different related work concerning the self-x properties of a system. The next section will provide a comprehensive list of assumptions, restrictions and considerations which are derived from the domain analysis that has been provided so far. The given aspects present the constraints for the investigated approach in the context of this work.

3.5 Differences between Related Work and Proposed Approach

This section will outline the difference between the related work and the proposed approach within this thesis. A short recap about the introduced related work will be provided in the following paragraphs. Each paragraph will close with a delimitation to the present thesis. The remainder of this section will outline the several assumptions, restrictions, requirements and design considerations which are considered for the design and implementation of the proposed approach.

The related work about operating systems that was presented in section 3.1 shows a great variety of possible approaches for supporting a flexible task management on a multi-core system. These operating systems support either a dynamic (real-time) workload or a mixed-critical workload. In spite of efforts for realizing an operating system which supports both workloads, it is a challenge to consolidate the different requirements into one unified system approach. Therefore, this thesis proposes a consolidated architecture that is designed to support dynamic mixed-critical workloads.

The scheduling mechanism of an operating system can be seen as a central part for supporting a dynamic mixed-critical workload (see section 3.2). Dynamic workloads are hereby supported by the combination of several, but different, scheduling strategies to a more flexible scheduling approach which can also be extended by an online schedulability analysis. In case of a multi-core support, scheduling strategies are combined with online assignment or allocation mechanisms. For supporting mixed-criticality workloads, there are efforts to extend the task model of a scheduling algorithm. Summarizing all efforts, a support for dynamic mixed-critical workloads thus introduces more complexity in the actual scheduling approach. The proposed management design thus foresees a separation between the actual flexible scheduling (enforcement) and the admission or allocation mechanism. Moreover, both enforcement and admission are designed to be executed on the same device. This thesis proposes an operating system design which supports this separation on the same device.

In section 3.3 cited related work considers one of the main challenges within the context of mixed-criticality systems, namely resource management. A main problem, hereby, is partition (i.e. isolation) of the given resources by simultaneously sharing these

3 Domain Analysis

resources between mixed-critical software components for an efficient usage. Supporting dynamic mixed-critical workloads, however, enforces this problem and requires run-time control mechanisms. On the one hand, these mechanisms are placed directly within a scheduling strategy. On the other hand, the control mechanisms are realized as part of a hypervisor/operating system for multi-core systems. Contributing to an operating system support for dynamic mixed-critical workloads, this thesis will outline an approach which executes both the online control mechanism and the actual scheduling on the same device during run-time. Moreover, this work foresees the application of the proposed approach within a microkernel-based real-time operating system.

For designing a concrete control mechanism, the related work in section 3.5 outlines several approaches to manage complexity in a software-based system. A main result hereby is to grant the system more degrees of freedom by equally controlling the system behavior. The cited works for enabling self-adaptive system properties include concepts for control mechanism and possible architectures/paradigms. These concepts are also applied in the context of safety-critical systems. Thus an application in the domain of mixed-critical systems seems feasible. However, there is no concrete control mechanism which can be used in the context of dynamic mixed-critical system. In accordance to the mixed-critical control mechanisms, this thesis proposes a solution for a possible self-adaptive control mechanism for dynamic mixed-critical workloads.

The following lists will outline several assumptions and design considerations according to the remainder of this work. For designing a flexible task management, the combination of several aspects stemming from the different research domains is required.

Assumptions The following assumptions are set for the design approach within this work:

- An initial system configuration is executed on the hardware platform: The initial deployment of system relevant components is done. There is no intention for the investigation in offline partitioning, exploration and deployment algorithms in the context of this work.
- The used embedded hardware platform is able to run ordinary operating systems: The hardware platform allows e.g. virtual memory management and thus contains a memory management unit.
- The hardware is assumed to working correctly: A hardware fault will not be detected.
- The hardware platform is considered as COTS: There is no special hardware platform required other than available for customers.
- Runnable on Embedded Multi-Core Platforms: With the advance of multi-core systems in the embedded domain, it is affordable for the operating system to be able to manage the underlying cores.

3.5 Differences between Related Work and Proposed Approach

- Communication interface with other systems: As part of a greater environment, the operating system needs to provide corresponding interfaces for an external communication.
- Components, Tasks and Threads: It is considered, that components are single-threaded tasks. So, a possible lock-situation like deadlocks inside one task can be circumvented. Functional parallelism, and thus utilizing the underlying multi-core architecture, is achieved by execution of different tasks on different cores (multiprocessing).
- Cores are independent: several cores inside the system can be considered in an independent way. Each core with its scheduling scheme can be seen as an uni-processor system, where former invented algorithms are applicable.
- Focus on software controls rather than hardware: The proposed system with its flexible resource management is exclusively designed for software components. Software control mechanism (e.g. admission) therefore are clearly in the focus.

Restrictions The following aspects and their investigations are no subject for the developed design approach within this thesis:

- All Tasks are known in advance: The tasks and their information (e.g. prioritization, safety level etc.) used for the flexible task management are considered as given. There is no creation of new tasks during run-time.
- The components are binary compatible across platforms: The exchange of components works only between hardware platforms with the same architecture (e.g. ARM, x86).
- Concrete automotive functions are abstracted by artificial tasks: In the context of this work there are components labeled after known automotive components but are simplified versions.
- An external communication is assumed but only for testing: The required communication with other systems is merely used to exchange tasks/components. Neither a real-time communication nor a holistic approach are content of this work.
- Cores and criticality: Assignment by utilizing the mixed-critical classification of each task and each core. High-critical tasks are executed on high classified cores, whereas low-critical tasks are executed on low classified cores. Communication between cores is restricted under the assumption of equal criticality level.
- Execution of software run-time controls on the same device: The overall goal is to enable a local adaptation on the same device without the need of an external control unit. Receiving changes can be checked locally which reduces latency and network bandwidth.

Requirements The following requirements are mandatory for the proposed solution within this thesis:

- The hardware platform is at least a dual-core system: The intended approach for separation and overhead reduction is only feasible within a multi-core configuration.
- Small footprint (efficient resource usage): An operating system using immense capacities of the available resources by itself is not suitable for the embedded domain. Further, with regard to a real-time context, the operating system needs to form a small trusted computing base.
- Allowing of run-time flexibility and guarantee: Both aspects are mandatory to realize a flexible task management. Flexibility means the operating system is able to modify the working component during run-time. Many systems in real-time and mixed-criticality areas enforce the possibility of the used system to make guarantees about distinct system behavior.
- Sharing of local resources in a safe way: Important aspect for the usage in mixed-critical systems is the possibility of a fine-grained control about the resource sharing between software components. Corresponding mechanisms need to be available in the used operating system.
- Kernel-independent solution: Flexible management of resources should be independent from the underlying kernel as far as possible. This enables a complete application of the proposed concept. General idea is to implement a controller-based mechanism for the several strategies to manage the underlying structures like scheduler or thread-to-core binding. Also supporting components like synchronization or monitoring are proposed to be realized in this way.
- Criticality level and affinity (i.e. component assignment to hardware resources): Both concepts are similar but not equal. Scheduling mechanisms that handle these aspects are different. The concept of an “affinity” is mandatory for the system due to changing requirements. The configuration (i.e. which component runs on which core) needs to be flexible. Also the timing aspects and the resource sharing are affected by these differences. Flexible resource management needs to deal with this in a consistent way.
- Compensation of introduced overhead: By separating the adaptation logic from the system logic by using the proposed multi-core approach, the introduced overhead can be subsumed.
- Efficient controlling and monitoring methods: Providing the adaptation mechanism on the same embedded device requires an efficient implementation of adaptation methods.

3.5 Differences between Related Work and Proposed Approach

- Modification of software components at run-time: The proposed system is at least capable of adding and removing of components during run-time. A modification of component parameters (e.g. priority) is not intended so far.

Design Considerations The following points considers general directions for the design of the proposed approach:

- Allowing several scheduling policies running on the same system: The benefit of using a multi-core based system is not to increase performance rather than allow a heterogeneous partition. This is achieved by assigning different scheduling policies to distinct cores.
- Online analysis as preferred method: In the context of a flexible task management all analysis mechanisms which are required to realize the management should be on-line. Thus the system is able to react on dynamic contextual changes.
- Combination with admission tests: The dynamic adaptation of the system also requires the combination of scheduling (enforcement) and admission. The later is responsible to guarantee a valid system configuration where the enforcement mechanism is responsible for the execution. A separation of both mechanisms allows the combination of different algorithms.
- Improved robustness of single scheduling policies through adequate optimization: The used scheduling algorithms have their benefits and drawbacks (e.g. transient overload situations). A continuously running optimization mechanism as part of the admission grants the possibility to eliminate potential hazardous states.
- Scheduling and criticality: Due to the static scheduling decision, FPS is used in hard real-time scenarios where guarantees outshines flexibility. EDF, however, can be used as well in soft and hard real-time scenarios where in this thesis it is enabled for soft and quality assurance scheduling due to its great resource utilization. But, in some scenarios there is the possibility where neither priorities nor deadlines are sufficient. In this scenario a more adaptive scheme via value-based scheduling (VBS) need to be applied.
- Strict separation of components with different criticality: As a key concept of mixed-criticality systems, it is mandatory that components with different criticality level are separated to circumvent possible influences between those.
- At least support for two criticality level: There are many research approaches which investigate in more criticality levels but two level is rather more investigated. It is the least common base for design and development of mixed-criticality system.
- Separation of adaptation logic and system logic: Similar to the separation in mixed-criticality systems, adaptation logic needs to be separated from system logic. This is for reducing the introduced overhead by the adaptation mechanism execution

3 Domain Analysis

(e.g. permanent monitoring). A failure in adaptation logic does not influence the system logic (i.e. no adaptation possible but system stays functional).

Conclusion

This chapter has provided an overview about the related work in the fields of operating systems, real-time scheduling, and system design properties to realize a flexible software management solution within a mixed-critical system context. The chapter has been closed by a section about distinct attributes which are (e.g. assumptions, restrictions etc.) combining the related work with the own thesis. The mentioned requirements and design considerations will be the starting point for the following chapters. Based on the investigations from this chapter, chapter 4 will therefore describe the author's architecture approach to combine flexible software management and mixed-criticality system attributes.

4 Design of a Consolidated Self-Adaptive Software Architecture for Multicore Systems

This chapter will propose an operating system design for an embedded mixed-critical multi-core system (cf. contributions 2 a,b). The design foresees that applications can be updated during run-time in respect to the chapter 3 outlined requirements and design considerations. Beginning with an outline of the overall architecture, a short description of relevant hardware/software components will be given in section 4.1. Further, the interconnection of the outlined software components as well as their functionality within the system will be presented in section 4.2. For the integration of specialized software components (i.e. controller), a possible allocation of software components to the underlying hardware platform follows in section 4.3. With the outlined software components in place, section 4.4 will describe possible control flows supported by the design.

4.1 Description of the Overall System Architecture

This section will introduce a system architecture which is designed according to the findings in section 3.5. The overall architecture design can be seen in figure 4.1 where the *System* is embedded in a greater network (i.e. *Environment*). In the following, the architecture describes exactly one system within a network.

The available hardware resources of the system are provided by the *Hardware Platform* which resides on the bottom of the system. In detail, the following physical system resources are considered:

- *Central Processing Unit* (CPU): homogeneous multi-core configuration for executing threads
- *Input/Output Devices*: sensors and actuators
- *Memory*: main memory (i.e. random access memory)
- *Communication Interface*: bus-based network communication

Altogether, the *Operating System* represents the system software which is executed on top of the hardware platform. The operating system is hereby divided in a *Kernel Space* and a *User Space*. The purpose of the kernel space is to provide an abstraction (i.e. kernel objects) from the underlying physical hardware resources to the software components

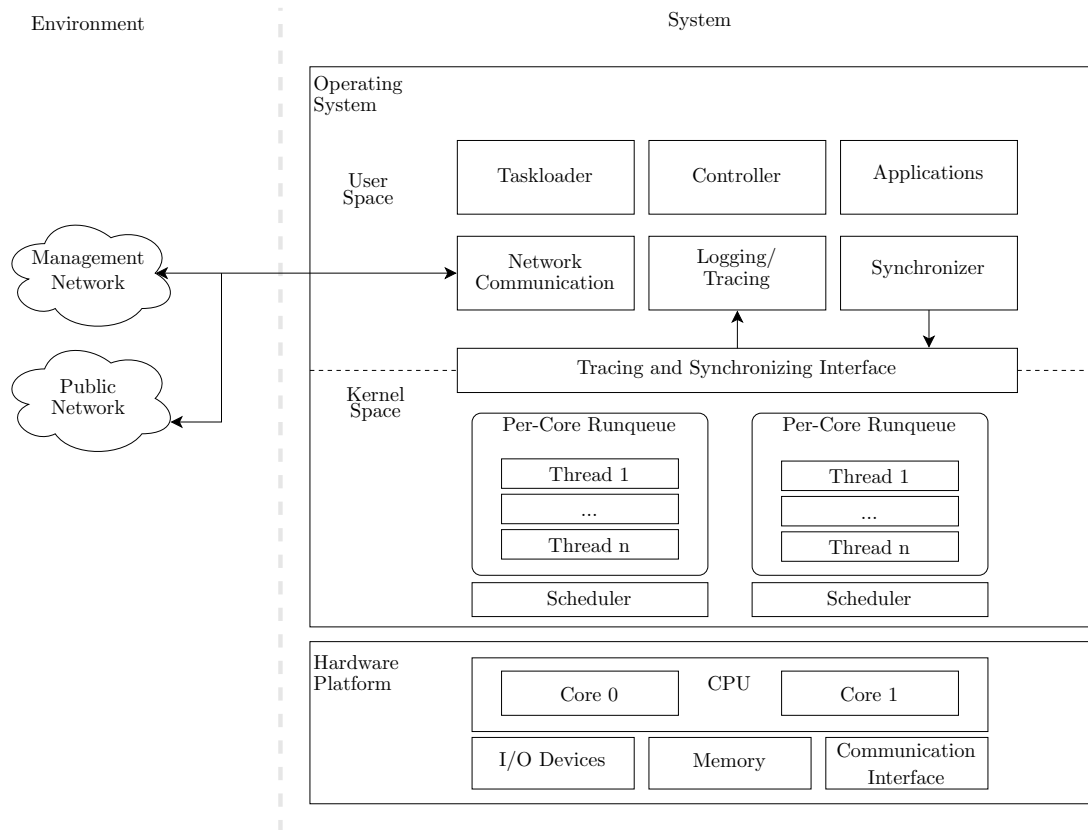


Figure 4.1: Block Diagram of the Overall Task Management Architecture

executed in user space. Within the user space, several software components are realizing the flexible task management. Both spaces are allowed to interchange information via a *Tracing and Synchronization Interface*. More details about the function of each kernel object and software component will be given in the following.

The kernel space provides the following kernel objects and internal data structures:

- *(Thread) Scheduler*: manages the execution order of threads by using a distinct queue-based scheduling strategy
- *Thread*: represents the execution of a software component
- *Run queue*: represents a queue per core with runnable threads

Scheduler The central role of the kernel thread scheduler object is the management of the underlying central processing unit and its cores by assigning CPU time to different execution threads. In general, the scheduling component consists of data structures (e.g. queue), strategies and interfaces related to the cpu management according a partitioned scheduling scheme. The dynamic queue-based data structure (i.e. changeable

during run-time) consists of several execution threads which need to be executed on the cpu. Each scheduling strategy is working on these queues. For a flexible task management, the kernel thread scheduler is therefore a crucial kernel object to get per-core information about the overall state of the ready queue and single threads.

The software components within the user space can be divided in:

- management components
 - *Controller*: (re-)configuration of new ready queue
 - *Synchronizer*: deploying of ready queue
 - *Taskloader*: loading of new applications in form of tasks
- common purpose components
 - *Network Communication*: communication with the environment
 - *Logging/Tracing*: gathering and parsing of operating system information
- application components (*Applications*): varying functionality and complexity

Controller In general, the controller component constrains the system adaptation (e.g. new system state) depending on the given instructions for creating a new task or optimizing the system according to a selected target function (e.g. utilization). The controller component hereby contains different processes for analyzing and planning of new system states. Whereas, the analysis process depends on services provided by the monitoring component about a current system state. The planning process uses the results of the analysis process to calculate a valid system state with the demanded modifications and submits the result via a shared memory provided by the synchronization component.

Tracing/Logging The tracing/logging component is the central monitoring part of the flexible task management architecture and responsible for the collection, pre-analysis and aggregation of information from several sources within the operating system.

Taskloader For the dynamic deployment of new applications within the system, the taskloader component provides the relevant services to the network component. Whereas, the taskloader component is responsible for the management of its applications (e.g. starting and stopping). Hereby, it reads the provided meta information (i.e. task description) and creates a task (representation of an application) accordingly. Moreover, the taskloader component provides interfaces for the network component as well as for application subsystem. It is capable of adding and removing of tasks (for the network component) and starting and stopping of tasks (for the application subsystem).

Synchronizer The synchronizer component is used to keep the user space and kernel space in sync. It therefore calculates an adequate point in time when an update of the underlying kernel space (i.e. ready queues) is feasible. For this calculation it follows distinct strategies according to the state of the targeting ready queue which can be either empty or filled.

Network Communication The network component is responsible to communicate with a systems environment (e.g. other systems) and therefore realizes a server which diverts the internal services to external calls. Where most of the provided services affect the aforementioned components (e.g. monitoring, controller). The data exchange format used for the communication with other systems within the environment is file-based.

Both common purpose components and management components are invariant and hence unaffected from run-time adaptation. Whereby, not every common purpose component is mandatory in every adaptation use-case (see section 4.4). Basically, common purpose components realize the basic infrastructure for system modifications from the outside. Application components are subject to a run-time adaptation and can vary in functionality and complexity. For example, the functionality and complexity can range from sensor/actuator components over automotive specific components (e.g. adaptive cruise controller) up to virtualized components (e.g. infotainment, real-time system). Actually, a components functionality and complexity and therefore its system requirements determine the possibility to be a subject of an adaptation. In case of an adaptation, application components are allowed to be added, removed or modified for the sake of optimization and fail-safe state. But there are also application components which are vital for the overall system and where even a modification is risky (e.g. critical components). In this case the application component will be excluded from an adaptation. For a clear separation, the architecture utilizes the underlying multi-core hardware platform to allocate corresponding application components according to their adaptation capabilities (see section 4.3). So, individual application components are largely unaffected from a given adaptation process if required.

This section has introduced an architecture design supporting a flexible task management within a multi-core embedded system. All relevant components, objects and resources has been identified and explained. For answering how a component performs its actual functionality, further details about the insights of a component will be outlined in the next section.

4.2 Kernel Space and User Space Interactions

This section will provide an overview about the interconnections between user space software components among each other as well as to the underlying kernel space with its kernel objects. Based on an overall view of all connections within the operating system, the consecutive subsections will explain the involved kernel objects and software components in more detail.

The component and connectors diagram, which can be seen in figure 4.2, shows an excerpt of all mandatory components and their interconnections. Each component is equipped with distinct connectors symbolizing their service demand or service supply. The components hereby follow the naming convention of the already introduced software components with one exception: for describing the *Tracing and Synchronization Interface*, the kernel is assumed as a further component rather than a space.

4.2 Kernel Space and User Space Interactions

The information exchange between kernel space and user space includes information about the current system state which are provided mainly by the scheduling kernel object and are consumed by the tracing/logging component in user space (i.e. *system information*). On the other hand, modified system states are *deployed* by an user space synchronizer component to the underlying scheduling kernel object.

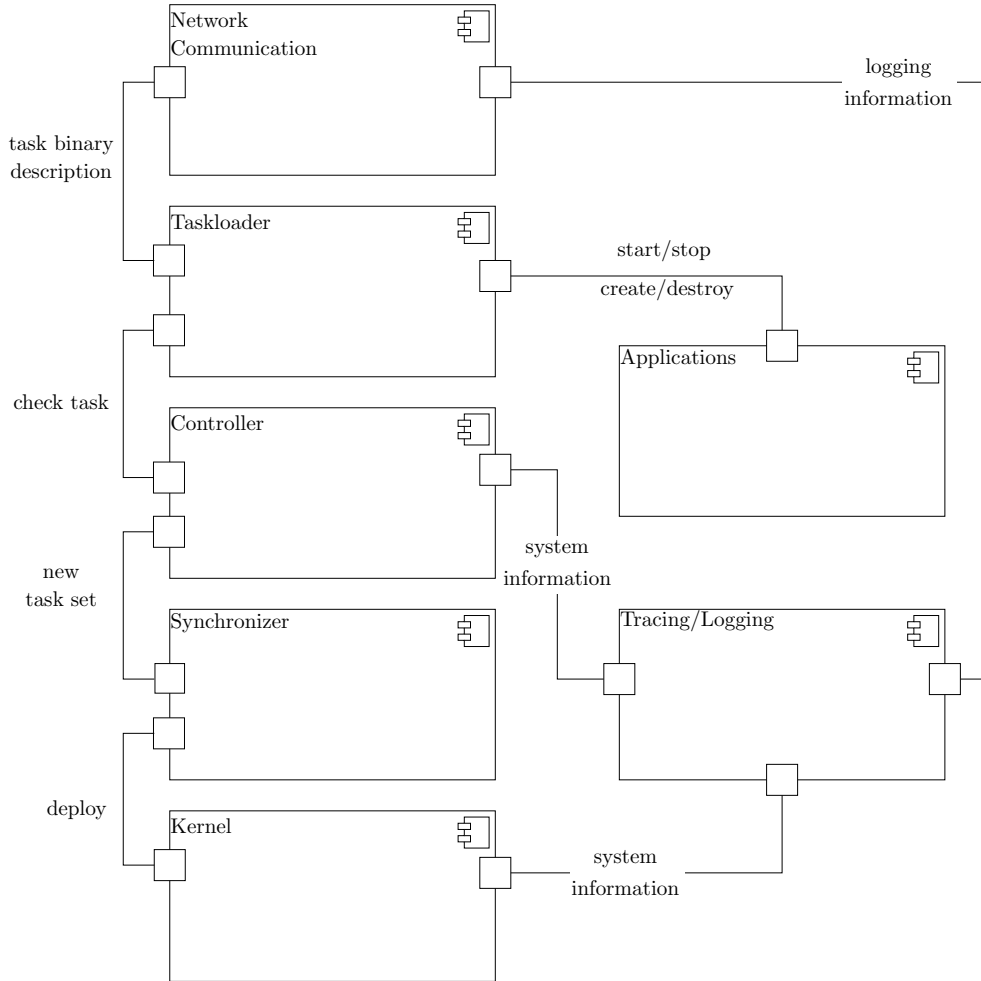
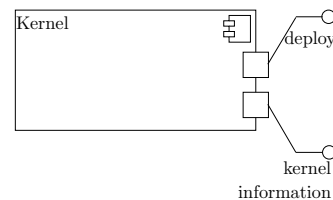


Figure 4.2: Mandatory components and their connectors

The kernel space provides two interfaces which handle the corresponding requests to get *system information* and to *deploy* a new ready queue.

Within the kernel space there are co-existing scheduler kernel objects which are assigned to the individual cores of the hardware platform. Each scheduler kernel object provides interfaces to modify its complete ready queue or one single thread. By calling the *deploy* interface, the task or ready queue is forwarded to the targeting scheduling object



by an internal routing mechanism. By calling the *system information* interface, the kernel collects the individual information of all scheduler kernel objects, their ready queues, and their running threads. These information are gathered from every individual kernel object (e.g. execution time from thread) and are bundled on a per-core base. In a last step these information can be combined to get information about either the complete kernel space or a single core.

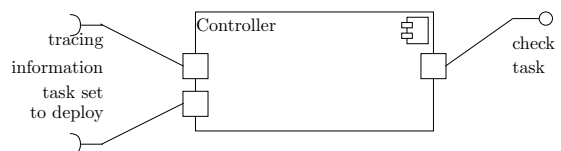
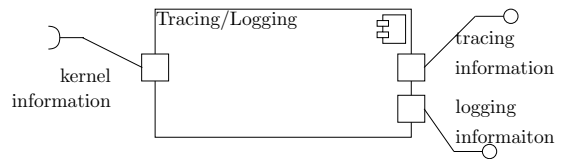
For providing system information, the tracing/logging component needs an interface for gathering the demanded information from the underlying kernel. Accordingly, the tracing/logging component serves these information via distinct interfaces to other user-space components. The tracing/logging software component thus provides the following interfaces: *tracing information*, *logging information* and *kernel information*.

The core of the collection routine which gathers the information about the kernel (i.e. via *kernel information* interface) is based on a tracing process inside the tracing/logging component. After the tracing, the following aggregation step combines these information to a common data

set (i.e. system state) which can be queried by internal other user-space components (e.g. controller) via the provided *tracing information* interface. The tracing/logging component hereby provides an unified interface which can be used to get the current system information. A call of the tracing interface, however, does not return the system information directly but rather as a reference on a shared memory region. This allows the sharing of system information between components by simultaneously reducing the processing overhead in the tracing/logging component induced by components concurrently requesting the current system information. The *logging information* interface includes a further processing step where the common data set is serialized for a later network transfer.

For controlling the adaptation of the system, the controller component includes a decision making process which is based on the current system information, the relevant task set, and the task (i.e. application) which will be subject to the adaptation. The controller software component correspondingly provides the following interfaces: *tracing information*, *task set to deploy* and *check task*. The system information which is requested via the *tracing information* interface contains the current ready queue of the target core. For a given application, the *check task* interface provides a test if the application can be enqueued in the current ready queue. In case of a positive result, the new ready queue is served via the *task set to deploy* interface.

Considering the co-existent scheduling strategies within the system, the internal structure of the controller component consists of several methods for analyzing the related ready queues of each core. In any case, an analysis leads to concrete instructions how the ready queues can be modified accordingly or not. Depending on the prop-

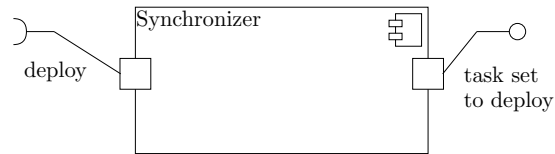


4.2 Kernel Space and User Space Interactions

erties of a given application, the controller decides which scheduling strategy on which core is capable of executing this application. Corresponding to the scheduling strategy, a analysis method will be chosen. There are also cases where a modification of a ready queue is not caused by the deployment of a new application (i.e. installation). The controller thus provides analysis methods for improving the overall system reliability by optimizing existing ready queues.

The synchronizer software component deploys a given task set (i.e. ready queue) at a distinct point in time (i.e. reliability of the kernel can be guaranteed) to the underlying kernel by calling its corresponding system calls. For providing this functionality, the synchronizer component possesses the interfaces *deploy* and *task set to deploy*.

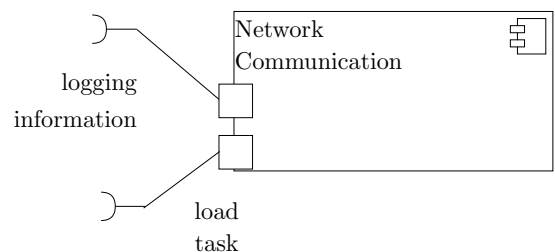
For transferring the new task set from the controller component to the synchronizer component, the *task set to deploy* interface will be used. Within the synchronizer component, a distinct point in time for deploying the given task set to the



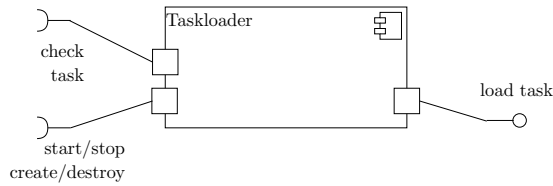
underlying kernel will be calculated. The calculation of a suitable point in time depends, therefore, on the state of the kernel queues which are either empty or filled. The sync of an empty queue is relatively straightforward and not bound to special constraints. Whereas, for a filled queue there are additional synchronization points possible, namely fixed switching point and variable fixing point. For the former case a dedicated switching task marking a save point for synchronization (e.g. idle task) can be used. However, for the latter case of variable point in time switching, a more complex routine needs to be elaborated due to difficulties arising (e.g. task dependencies). In the end, the given task set will be deployed to the underlying kernel via the *deploy* interface. This can be done in two distinct ways: as a complete ready queue or as task by task. In both cases the kernel's system calls are used.

The network communication component is used to get applications in the system and to get logging information out of the system. For this purpose, the network communication component uses two internal interfaces, namely *logging information* and *load task*.

The network communication component provides external interfaces for communicating with the management network and the public network which are used to receive corresponding requests and data. To realize this functionality, the internal structure of the network component adapts a socket server. For serving the incoming requests, the services of a distinct user-space software component will be used. For example, the process of loading a new task utilizes the *load task* service of the taskloader component.



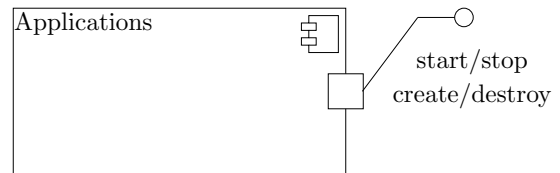
The taskloader component is the entry point of the flexible task management. Therefore, it provides interfaces for managing the applications. The following interfaces are provided: *check task*, *load task* and *start/stop create/destroy*.



The internal structure is based on a task set storing the applications which are managed by the flexible task management. A new task is transferred from the network communication component to the task loader via the *load task* interface. After that, the taskloader component uses the *check task* interface of the controller to check which modifications on the current task set are mandatory. Related applications can be managed by the *start/stop create/destroy* interface. For concrete workflows which demonstrate the interplay of the taskloader component, controller component etc., see section 4.4.

An application component is managed via the taskloader through the *start/stop create/destroy* interface.

An application component hereby consists of a (meta) description, monitoring data and the actual binary. The description as well as the monitoring data is required to manage the application and provides information about its current state. During execution, this is important to determine if the actual application execution (i.e. traced by the monitoring component) is compliant to the given specification (i.e. task description). This reasoning embodies the foundation of the decision process within the controller component. If an application gets created by the taskloader, the description is used to initialize the timing aspects of the binary (e.g. period). Furthermore, the application code in binary form is loaded into the application. The binary contains an execution thread which is consumed by the underlying scheduler kernel object. Starting the application formally starts the execution of the thread.



This section has described the interactions between the user space software components. Therefore, distinct interfaces of each software component were presented.

4.3 Allocation of Software Components to Multicore Hardware Platform

In this section, an allocation between software components and hardware platform resources (i.e. cores) will be developed. Hereby, functional and dependability requirements of software components will be addressed. As a functional aspect, applications should, as far as possible, not be influenced by the execution of the remaining software components. Possible deployment configurations therefore will be outlined in section 4.3.1. Dependability requirements, on the other side, have a strong relevance to the critical-

ity of a software component. For that reason, a criticality aware mapping of software components with possible isolation capabilities will be presented in section 4.3.2 and section 4.3.3.

4.3.1 Integration of Application Components within the System

This section will provide several deployment strategies for software components. A possible performance degradation of application components through the remaining system needs to be avoided. In general, the management and common purpose components are an attachment for the overall system functionality (i.e. provided by the application components) introducing overhead and should therefore, be separated from the application components. Moreover, in case of a failure within the management or common purpose components, a separation ensures the continuous execution of the applications (i.e. dependability). Indeed, there exists several strategies to place the application components within the overall system and although each strategy is fostering the reduction of the introduced overhead, they all come at a cost.

Within the described system architecture, application components can be principally integrated in either the user-space of the operating system or directly on top of the hardware platform (i.e. with or without a common software stack). By integrating an application on top of the hardware platform, the application will be executed on a distinct core and is separated from the remaining operating system. In the context of this section, this scenario will be referenced as *bare metal* integration. By integrating the application within the operating system (i.e. using a common software stack with the remaining software components), there are two scenarios conceivable: directly as application component within the user-space or indirectly as part of another application. The first scenario will be referenced as *native* integration. The second scenario will be referenced as *stacked* integration. An overview of the available integration scenarios is depicted in figure 4.3.

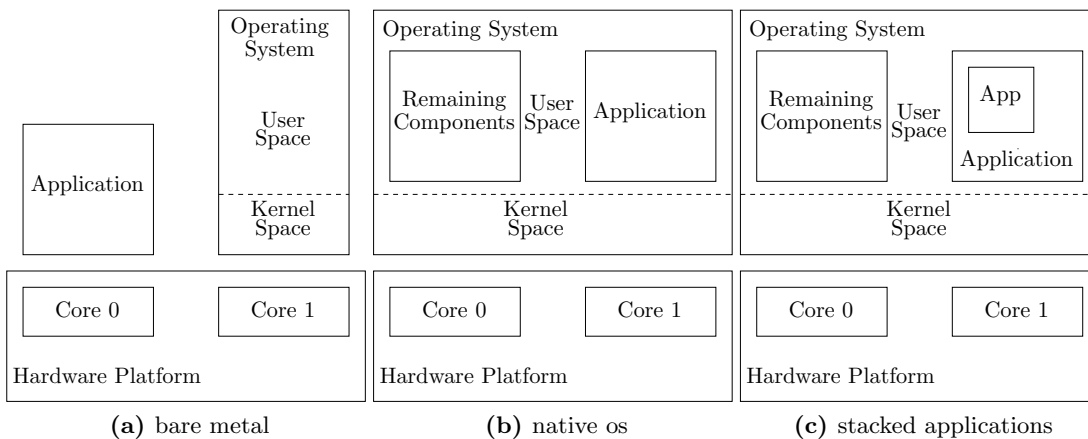


Figure 4.3: Several mapping strategies for application component

The deployment scenario as depicted in figure 4.3 (a) describes the integration of an application component directly on top of a dedicated processing core (i.e. bare metal) of the hardware platform. The application hereby is separated from the operating system completely. In this case, the remaining software components don't interfere with the application directly. This isolation can be used to implement an application which takes care of its own resource management. This leads to a number of configurations where the application can be a simple program, a complete operating system or a virtualized solution. Obviously, this allows a very flexible use of the application whereby services which are required for managing this application also need to be developed in conformity with the remaining system. According to the management, an application residing on one dedicated core need to utilize a dedicated communication mechanism to provide their services to the system software. Due to the fact that executing an application exclusively on a dedicated core, this scenario prohibits the usage of this core by other system software components (i.e. core is not managed by system software) which results in a waste of valuable resources. Considering the performance of the chosen deployment scenario, the execution of an application directly on a dedicated core embodies by far the greatest performance potential in comparison to other deployment scenarios. In this case, even applications with tight timing constraints could be integrated. For the availability of the system, an application which is deployed as depicted intersects the system software architecture which has a questionable impact for the stability of the overall system if the application will fail. In summary, executing an application on one dedicated hardware platform core results in high performance regarding computation speed (fast). On the other side, the strict separation from the rest of the system software complicates the management of resources and dependability related aspects.

The native deployment scenario where the applications are part of the operating system user space offers some benefits regarding implementation, management, performance and reliability. A detailed overview of the scenario is depicted in figure 4.3 (b) where the applications are able to use the services and resources from the software stack. By using this software stack, the applications are able to use common services already provided by the operating system. This reduces the implementation overhead in comparison to a bare metal solution where even basic functionality need to be implemented. In this case, the development of an application also can be focused on the essential functionality. The management of applications as part of the operating system on top of a homogeneous software stack allows the consistent usage of the systems infrastructure for communication and processing (e.g. scheduler). A seamless integration of the applications is thus feasible, where existing communication mechanisms can be utilized to further reduce the overhead. Moreover, as part of the operating system, applications can be represented by the same task model as remaining components within the system which eases the timing analysis (e.g. calculation of the introduced overhead). Considering the performance, applications can also be separated from the remaining software components by deploying the applications on other cores. Of course, the used software stack needs to be flexible enough to provide this deployment on several cores (e.g. affinity). Furthermore, the software system stack needs to allow a fix binding of applications to corresponding cores at run-time. Regarding the dependability of the overall system, a failing application can

be restarted by the established management mechanism. So, the outlined deployment scenario achieves a balance in the areas of implementation, management, performance and reliability. After describing the first deployment scenario with a system software stack, in the following an alternative deployment will be outlined.

The deployment scenario shown in figure 4.3 (c) describes a further option to integrate the applications on top of a system software stack. An application can be executed as part of another application which is a native operating system component. This allows the stacking of applications. In general, this approach introduces a greater abstraction for the integration of an application apart from the operating system. This enables a greater software-based separation of the application from the remaining components of the operating system even if it's a part of it (i.e. loose coupling). Considering the implementation of this deployment scenario, applications can also be part of a virtualized operating system (e.g. Linux) mirroring the design possibilities of the bare metal deployment scenario. This enables a great implementation flexibility due to the abstraction from the underlying operating system. Considering the management of such stacked applications, their surrounding applications are still managed by the operating system user space components. But, the additional abstraction enforces the fact that the underlying system is not aware about the insights of the application component itself which means a restriction in the management possibilities. Moreover, through the higher abstraction level, an application's performance may suffer from an exceeding lag which is caused by the extended call stack of a function from the application down to the hardware. Especially in cases where applications have higher timing demands, this scenario may not be sufficient. In comparison with the native operating system deployment scenario, this deployment scenario derives a comparable flexibility in case of an application malfunction. In conclusion, the deployment scenario where applications are integrated within another application allows a comparable flexibility to the native deployment scenario considering the implementation and the dependability. However, the described scenario falls short in cases of management and performance where an additional abstraction on the software system stack introduces further barriers.

To sum it up, the provided deployment scenarios for integrating an application within the system emphasize different aspects of implementation, management, performance and dependability. Whereby, the deployment of applications within the native operating system finds a balance between these aspects, the other provided deployment scenarios clearly prefer one of these aspects at the expense of other aspects. All described deployment scenarios however are based on the assumption that the applications have the same criticality and thus are able to be deployed on an arbitrary core on the hardware platform. The next section therefore will provide the second aspect of the allocation, namely criticality.

4.3.2 Criticality-Aware Allocation of Software Components

In this section, relevant design decisions to realize a possible criticality mapping will be presented. Thus, the representation of a criticality level across different system parts like cores, schedulers and software components will be described. Cores, for example, as

part of the hardware platform need to be classified according to a criticality level. For each criticality level, a dedicated scheduler within the kernel space is chosen to fulfill the required timing demands. Further, it is important to extend the description of software components (e.g. task and execution) in accordance to a given criticality level.

Assuming that the combination of several software components can have the following criticality classification: equal or mixed critical. There are at least two criticality levels needed to allow the discussion about a mixed critical combination of software components. Software components therefore are either low-critical (lo) or high-critical (hi). This criticality representation needs to be respected even on core level, where cores with a certain criticality certification can be expensive and part of a special hardware device. However, to bring the designed approach to a great number of general purpose platforms rather than few special purpose ones, this thesis focuses on an exclusively software-based realization of criticality classification on top of the hardware platform without specially certified cores.

For the management of available hardware platform cores by the kernel, the architecture approach foresees an allocation of a dedicated scheduler to each core. The criticality classification (e.g. high critical) and the timing behavior of a software component (e.g. hard real-time) are orthogonal¹. Both aspects of the software component therefore need to be correspondingly mapped to the available cores and the schedulers. For example, in case of a high critical component, the value of a software component could be defined as point in time where the result of this component is no longer relevant or harmful for the system. For that case, a deadline based scheduler (e.g. EDF) aware of handling deadlines could be used. Whereas, by non critical software components, fairness provided by the completely fair scheduler (CFS) could be demanded for keeping the timing behavior of a software component. According to the isolation of the resulting core set, each core is managed by one dedicated scheduler enabling a separation between software components running on other cores. This allows, but is not limited to, the realization of mixed-critical configurations of core sets where the set contains of one or more pure high-critical cores in combination with low or none-critical cores. Hereby, components for a low-critical core like the management components can be separated from high-critical application components by simply allocating them to one core and the application components to another one. Moreover, this approach allows the further usage of well tried single-core scheduling mechanisms in a multi-core environment, where the scheduling strategy on each core can range from simple to complex according to the criticality demands (i.e. even mixed-critical scheduler would be possible). In conclusion, there is a suitable scheduler managing each core in isolation from other cores as part of a multi-core environment. So, the next relevant allocation aspect address the admission of a software component to an available scheduler and the communication between cores to allow the cooperation between software components on dedicated cores.

The admission of a software component to a corresponding hardware platform core within the system software stack requires the usage of criticality, schedule type and core affinity which are part of the software components task description. Whereby, the

¹There are hard real-time software components which are not highly critical and vice versa.

criticality property and the scheduler type are both mandatory for a proper admission (see former paragraph). For example, assuming a deadline-based low critical software component: If the admission of this component takes place considering the criticality only, a low critical core with a priority-based scheduling policy (e.g. FPP) is also a valid admission target, hence, unable to deal with deadlines. Otherwise, with an admission considering only the scheduling type, a high critical core could be selected where a dynamic admission on this core can result in a hazardous system state. So, both properties, criticality as well as scheduler type are required for a proper admission whereas the chosen example is simplifying the available parameter combination. In general, the admission is based on the combination of all parameters describing the scheduler and core properties respectively. But, there is a problem in case more than one scheduler or core have the same capability to satisfy the admission demand for a software component. In case of scheduling type, there are more schedulers capable of dealing with deadlines (e.g. EDF, LLF) where a immediate solution is to specify the concrete scheduling policy rather than its scheduling aspect (i.e. deadline). Furthermore, there can be equal critical cores which are managed by equal scheduling policies (e.g. EDF) where both cores would be feasible for an admission. If the core binding is irrelevant for the admission of the software component, there is a certain flexibility for the admission mechanism to select one of the cores. In contrast, where the binding is relevant for the proper execution of a software component, there exists an additional affinity parameter which allows to guarantee that the software component is always executed on the selected core. In all cases, the admission mechanism is responsible to correctly enqueue the software component within the scheduler. The considered concept, so far, is based on the idea of separating (i.e. temporal isolation) the execution of software components on different cores according to their criticality classification. But, in case of management components and their to be managed applications, an inter-core communication mechanism is required.

The criticality classification of a software component also influences the inter-core communication and requires a decision which software components on different cores (with or without varying criticality classification) are allowed to communicate with each other². For example, management components (running on one core) require the communication with their to be controlled applications which are executed on another core. In combination with a corresponding criticality classification, the question arises: what is an allowed communication configuration for the relationship between management components and the to be managed components? In general, management components need the same criticality classification as their to be managed components. For example, a high critical classified management component is allowed to manage high critical application components. Whereas a low critical management component is responsible for low critical applications respectively. However under the assumption that a high critical component enforces a tighter timing behavior than a low critical component, there exists the possibility to control a low critical application by a high critical component but not other way around. So, an allowed communication configuration which can be seen as

²software components running on the same core are always allowed to communicate with each other if required

valid respects these criticality relations. Indeed, the criticality classification influences the allowed communication configuration but the criticality compliance also requires the technical support by the hardware platform (see section 4.3.3). The so far described design considerations for an allocation of a software component to the underlying hardware platform will be combined to an overall task management architecture in the following.

The architecture configuration given in figure 4.4 follows the so far described aspects of a critical aware allocation of software components. Whereby, the architecture is layered on the horizontal dimension according to the described operating system and hardware platform. Corresponding to this, the vertical dimension shows the system software stack on top of each hardware platform core. Each core with its associated software stack which is composed of software components is colored according to a certain criticality classification. Beginning with the individual software stack on each core, kernel objects are colored in green. Additionally, operating system software components are colored in orange. In this configuration, the application components (colored in blue) are part of the operating system (native deployment scenario) but mainly separated from the remaining management and common purpose components (colored in orange). In this configuration low-critical cores are colored in yellow and high critical cores are colored in red. In general, the chosen configuration allows the execution of application components without the impact of a possible adaptation process due to the fact that the controller component is not executed on the same core as the application component. This configuration, indeed, is one possible solution to the outlined allocation problem but rather simplified.

In conclusion, this section has described the several aspects of a criticality aware allocation of software components to the hardware platform. The allocation of software components (operating system/applications) requires both scheduling type and criticality classification where in some cases an additional core affinity can be helpful. Furthermore, the inter-core communication between mixed-critical cores is required to allow the management of applications by the management components running each on distinct cores. However, the partitioning and separation of software components according to their criticality makes no assumptions about the isolated concurrent execution of software components on the same system. That means, software components are able to influence or disturb other software components in their execution leading to unforeseen system states. The next section therefore presents another aspect which is also related to criticality, namely isolation.

4.3.3 Isolation Supporting Criticality

This section will describe the two isolation concepts which can be divided in this work, namely temporal isolation and performance isolation. Whereby, temporal isolation can be seen as a software-based mechanism for separating high critical components from low critical components. On the opposite, performance isolation is a hardware-supported mechanism which supports the additional separation of hardware resources like caches. Possible scenarios will be described where either a performance isolation or temporal isolation can be used.

4.3 Allocation of Software Components to Multicore Hardware Platform

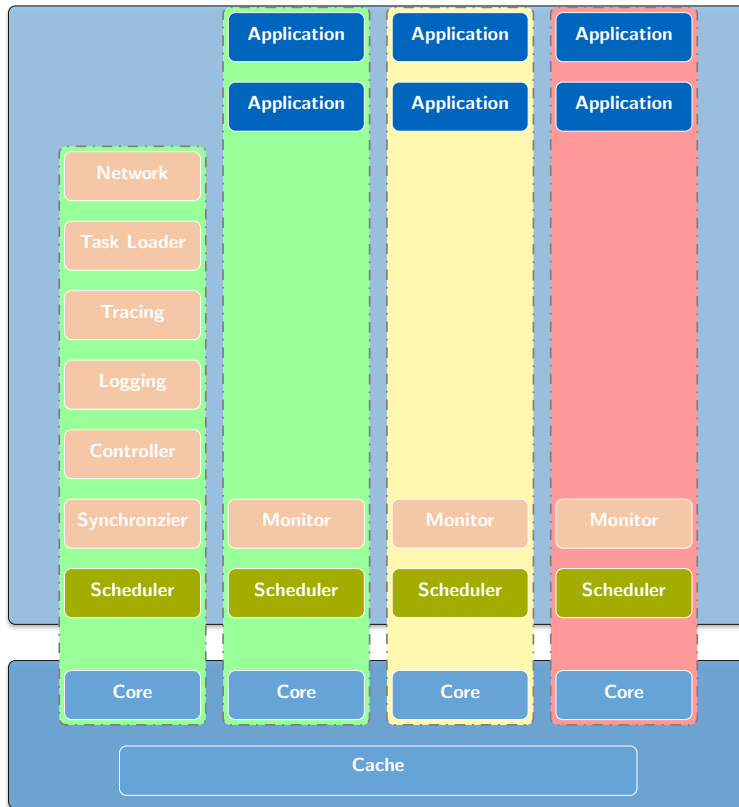


Figure 4.4: Allocation Example following the Partition Process

Temporal isolation, as depicted in figure 4.5a, can be achieved by the deployment of a high critical component on a dedicated core without the common system software stack (cf. bare metal deployment scenario). Hereby, high critical components are able to be executed independently from the remaining software components running within the system software. However, the software based separation of high critical components doesn't enforce the independent execution of software components as a whole. The reason for this are shared resources like caches. Malicious software components are able to utilize the shared resources to influence the timing demands of high critical software components on the other core. For that scenario, a software-based isolation is limited in case of hardware side attacks. That's why the concept of isolation need to be supported by hardware-based design concepts to circumvent the described scenario.

The separation of hardware-related parts like caches from remaining resources provided by the hardware platform is a essential design decision for the realization of the performance isolation. Hereby the former limited isolation approach can be extended for guaranteeing strong timing conditions. This approach is depicted in figure 4.5b where the isolated application has an exclusive access to core and its related cache. This system in system approach allows the integration of hard real-time components next to the software stack. Different variations are thereby conceivable: a real-time operating system,

a hardware emulation for high-speed calculations. Considering the communication between the completely isolated core with the other components within the system, a bus infrastructure could be used. However, performance isolation comes to the cost of higher management effort [47]. In conclusion, the combination of best effort components with high critical components within one system requires the usage of performance isolation for strong timing guarantees where software based temporal isolation is not enough.

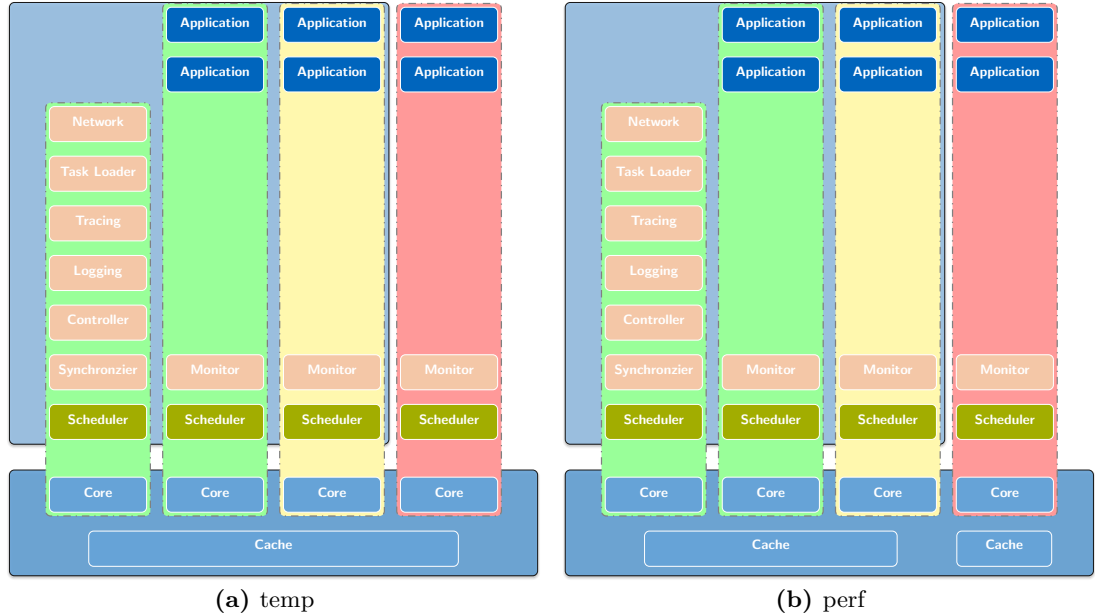


Figure 4.5: Temporal Isolation and Performance Isolation Approaches

To sum it up, this section has described an allocation of software components to the hardware platform. The section has described the separation of applications from the remaining software components. This separation considers that the underlying cores are equal critical which is not true in all cases. Thus, a criticality aware mapping of software components to the underlying hardware platform was covered afterwards. An important aspect hereby is the representation of criticality within the system. As a last step, additional isolation concepts for supporting the assignment strategy were outlined. The next section will describe the concrete adaptation workflows (i.e. dynamic behavior) by using the introduced software components.

4.4 Description of Workflows for Self-Adaptive Software Architecture Changes

The previous sections have been used to identify relevant software components, their connections with each other as well as their allocation to the underlying multi-core platform. What is missing so far is a concrete sequence for managing applications where

the management components and common purpose components are used together. This section therefore will outline three concrete workflows demonstrating such sequences. The section 4.4.1 will describe a sequence if a new application needs to be added to the system (i.e. install, update). The subsequent section 4.4.2 will focus on an optimization of the system to keep it reliable. The last section 4.4.3 will outline a possible sequence which can be used for diagnosing the running system by an external device.

4.4.1 Adding a New Task into the System

The figure 4.6 depicted activity diagram describes an adaptation workflow, where a new application component (i.e. task) is added to the overall system. During this flow, there are several processing steps required to actually deploy the new application into the system. Supporting these steps, an application consists of a *description* and the actual *binary*. An application description represents certain information about the to be integrated component like its execution (e.g. timing aspects) requirements. An application's binary represents the actual execution context of a component. The provided information is processed by the presented components during several workflow stages.

Starting with the network communication component, the transferred information about the component as well as their execution binary will be received. As a consecutive step, the network communication component transfers the received data and binary to the task loader component which reads the task description and stores the task binary accordingly. The task loader component keeps a set of applications which need to be managed. Each application which is allowed to be executed on the system will be stored within this set. As a first step, the task loader reads the provided component description and starts to constructs an application component considering the provided information. During this construction phase, the task loader emits the controller component to check if the application will fit into the system or not. This decision, if a application fits into the system, is made by a admission process within the controller component. For the admission process, the controller component requests the system information from the tracing/logging component and will update its internal information about the system respectively. Both the updated system information and the task are required for deciding if a deployment of the task into the system is feasible or not. In case of a negative admission result, the workflow terminates and the task construction is aborted. If the new application can be successfully deployed into the system, the task loader gets positive feedback from the controller and stores the allowed application in its set of applications. Moreover, the synchronizer component will get informed about a required update. The remaining workflow steps include the update of the ready queue as well as its deployment by the controller and synchronizer component together. After receiving the new ready queue, the synchronizer calculates an adequate point in time when a synchronization with the kernel is possible. Finally, the synchronizer component deploys the new ready queue to the kernel by calling the relevant enqueue/dequeue system calls.

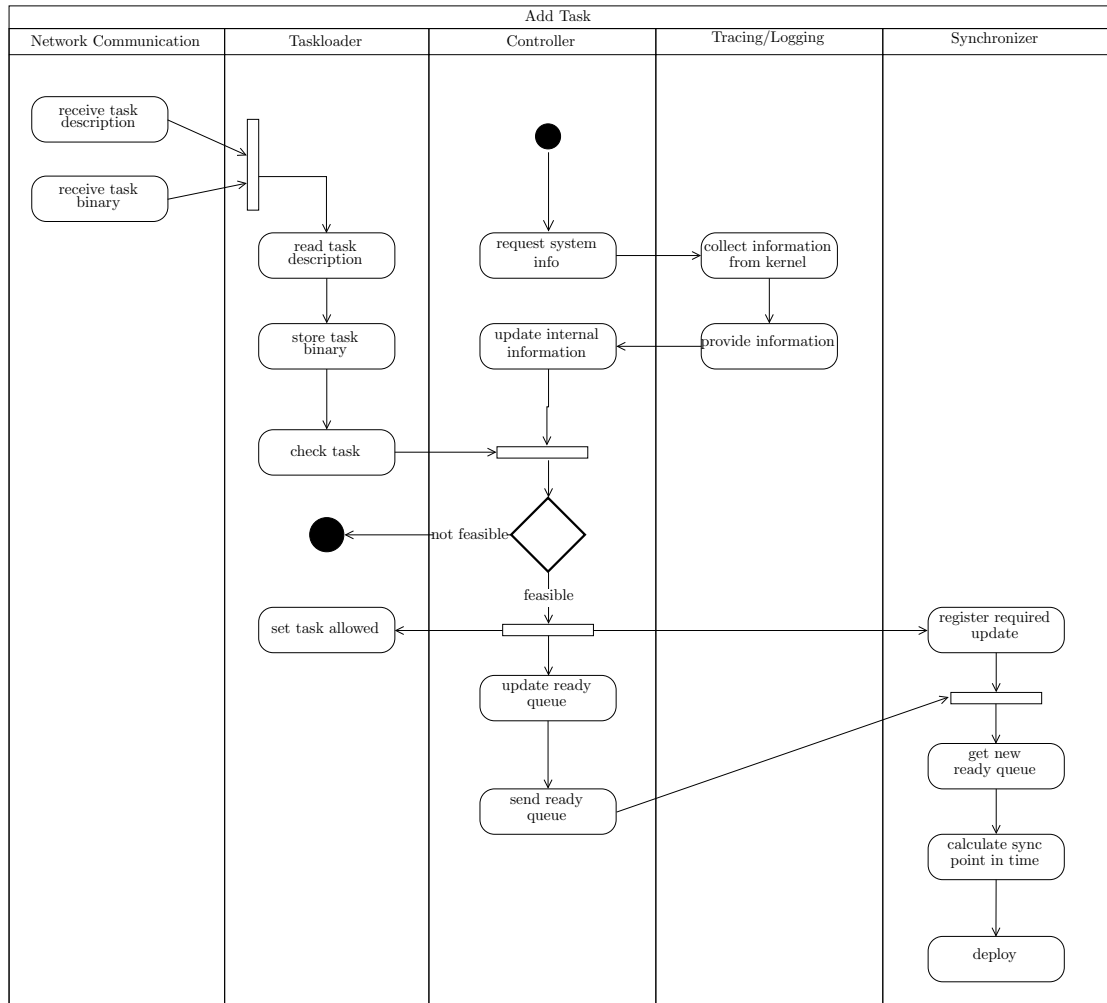


Figure 4.6: Activity diagram for inserting a new task within the system

4.4.2 Optimizing current System State

The activity diagram which is shown in figure 4.7 describes an adaptation scenario where a given system state will be optimized to e.g. avoid a potential hazardous system state. In contrast to the described scenario of adding a task, an adaptation process is started by an internal change within the system like finishing an application. Starting within the tracing/logging component which is continuously monitoring the system, a change in the execution behavior of the system will be observed. After that, the controller component will request the new current system information which is provided by the tracing/logging component. This topical information is used to update the internal information within the controller component. According to a given optimization goal, the controller component starts to optimize the current system state. The controller therefore decides which tasks are allowed to be executed and which tasks are not. The

4.4 Description of Workflows for Self-Adaptive Software Architecture Changes

result of the optimization step is sent to the task loader component and the synchronizer component. The task loader component updates its set of applications considering the result of the optimization process. The remaining steps for getting the new ready queue to the kernel are similar to the steps which were described in the former scenario. As before, the resulting ready queue is handed over to the synchronization component. Again, the synchronizer component will calculate an appropriate point in time where the synchronization can take place.

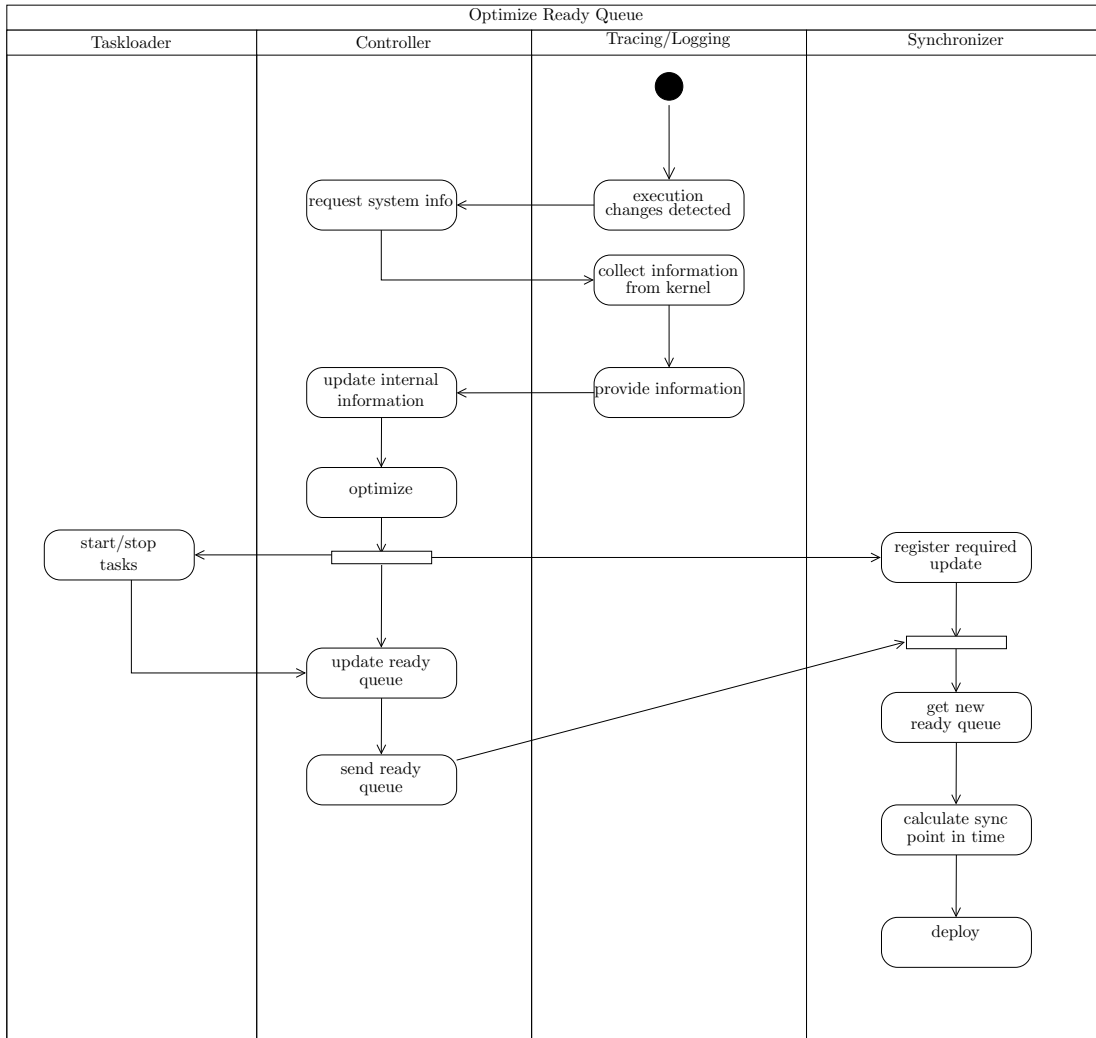


Figure 4.7: Activity diagram for optimizing a ready queue

4.4.3 Monitoring current System State

The activity diagram shown in figure 4.8 depicts the scenario when tracing or logging information about the current system state is delivered to an external system for e.g. di-

agnosis purposes. The workflow for the external request starts with the network communication component where a corresponding message will be received. The tracing/logging component collects the demanded information again from dedicated system calls provided by the kernel. In contrast to an internal usage of the collected information by other software components, an additional step is required which will prepare the information for a later transmission over the network. Within this step, the information will be serialized into a file-based format. After that, the network communication component gets the file-based description of the system state and sends it back to the other system.

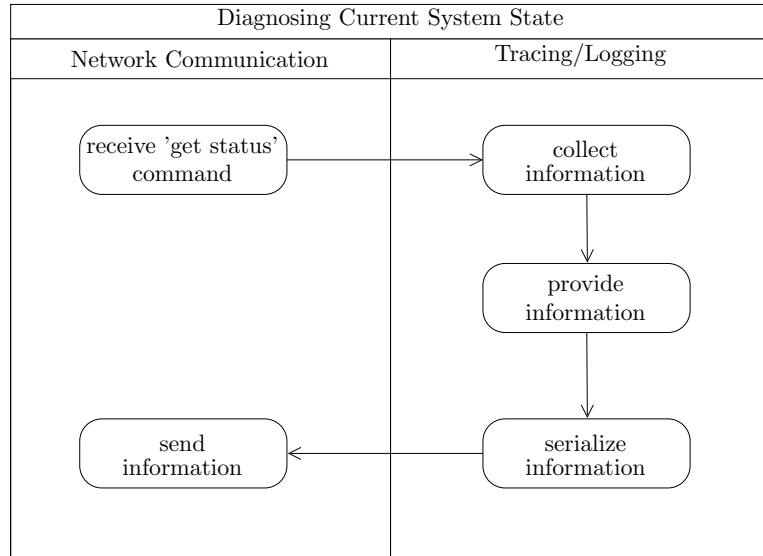


Figure 4.8: Activity diagram for diagnosing the current system state

This section has described three workflows for adapting the designed system architecture. In detail, activity diagrams for adding a new task, optimizing a given system state and sending system information for diagnostic purposes were provided. Within these activity diagrams the run-time activities of each software component were outlined.

Conclusion

This chapter has provided a consolidate architecture approach to integrate a flexible task management within a multi-core mixed-critical embedded system. The chapter therefore has started with an overview about the complete architecture. Within this section, relevant hardware resources, software components, and kernel objects as well as their distinct roles were identified. The subsequent section has described the interconnection of software components and kernel objects by providing the distinct interfaces and the internal workings of each software component. Further, the allocation of the software components to the underlying hardware platform were discussed. The focus hereby was on the functional separation of applications from the remaining components and a criticality aware allocation mechanism. After describing these aspects, the last section

4.4 Description of Workflows for Self-Adaptive Software Architecture Changes

has outlined the run-time behavior of the software components considering three different adaptation scenarios. As a next step the so far described design will be refined with concrete methods and algorithms to provide the postulated management mechanisms.

5 Flexible Task Management

This chapter will outline the concept of a flexible task management which is based on an adaptation process of a given software system (cp. contributions 3 b,c). The several steps within this adaptation process are subsumed to a consistent integration framework which is able to integrate mixed-criticality applications into a software system during run-time. A model, which will be described in section 5.1, represents the current running state of cores and ready queues. This model will serve as a base for the distinct reasoning steps within the integration framework. The current integration framework will be presented in section 5.2. Afterwards, section 5.3 will describe which software component is responsible for which processing step of the integration framework.

5.1 Basic Model for Reasoning about Current Running State

This section will describe a model which is used by the distinct integration framework steps to reason about the current running state of the underlying kernel space. This model is used during run-time and is stored on the same device. The design concept hereby foresees that the majority of the algorithms are executed in user-space which encapsulates the complexity from the kernel. With the usage of a multi-core system, a regeneration of task sets can be done on one dedicated core during run-time. The model, hereby, supports the representation of cores and ready queues within the user-space which will be described in section 5.1.1. Moreover, each application which needs to be integrated within the system consists of a description representing a combination of the required mixed-criticality parameters as well as the required timing parameters. The extended description will be outlined in section 5.1.2.

5.1.1 Ready-Queue and Core Representation within User Space

Due to the strict separation of admission and enforcement (i.e. integration framework and kernel scheduling object) in user-space and kernel-space, a direct access from user-space software components to the underlying ready queues is not allowed. Moreover, providing a representation of the underlying ready queues makes this approach kernel agnostic. Thus, the processing steps of the integration framework are done on a model-based representation of the underlying kernel space. This representation models the cores and the ready queues respectively where each core gets a ready queue assigned. A first design and implementation of this concept was demonstrated in the work of Nieleck [113].

A ready queue is represented by a *rq_buffer* containing the tasks that will be executed on the assigned core. For a fast access by the several integration steps, this object

should be shared between the distinct software components. The size of a *rq_buffer* can be arbitrarily set via the configuration phase of the system. Due to the fact of an embedded context, the maximum size of this buffer needs to be carefully chosen (e.g. 100 tasks). Beside the array storing the tasks, there are common methods required for the adding and removing of tasks to a *rq_buffer*.

Each core of the hardware platform is described by a *pcore* representation. A *pcore* gets a *rq_buffer* assigned. An important information that is related to a physical core is its criticality classification. The concept prohibits that tasks with different criticality classifications are executed on the same core. For an empty core, the classification of the first arriving tasks will be used to initialize the core classification. For example, a core is allowed to execute high-critical tasks if and only if the core itself is classified as high-critical. In contrast, if a core is classified as a low-critical core it is allowed to execute lo-critical tasks exclusively. *Pcores* stores these criticality information within a dedicated variable. Mapping the distinct states of a physical core, a *pcore* also supports the following states: active, standby, and off. A *pcore* is identified by an *id* which is represented by a unique number. Each *pcore* $C_i = (pol_i, id_i, \mu_i)$ defines the following information:

- $pol_i := FP, EDF$ defines the scheduling strategy which is used on this core
- id_i defines the identification number of a core
- μ_i defines the criticality of each core

After describing the basic representation of the underlying kernel space, the next section will cover the content of each *rq_buffer*, namely tasks. The task description which is used in a later reasoning within the integration framework will especially be into focus.

5.1.2 Extended Task Model Considering Criticality and Schedulability

Each software application component is represented by a task that resides in one of the user-space ready queues. This section will describe the extended task model considering the criticality properties and the timing properties of a task. The given (static) properties are used by the run-time integration framework during the dispatching and admission phases. Additionally, the description contains dynamic information about a task that is required for the later optimization steps where this information is used to decide if a task is allowed to be scheduled.

An application software component K_S is represented by a finite set of tasks $K_S := \tau_1, \tau_2, \dots, \tau_n$, where a task τ_i has the following static information:

- L_i defines the criticality classification (either *HI* or *LO*)
- T_i defines the minimal arrival time (period) in case of a periodic task
- D_i defines the absolute deadline

- C_i defines the computation time or Worst Case Execution Time (WCET)
- P_i defines the priority for priority based scheduler
- S_i defines the scheduling strategy which is required by this task
- A_i defines the affinity for manually assigning a task to a core

The so far observed task attributes are mostly fixed during the execution of a task. If an arriving task needs to be integrated into the system, an online admission/acceptance test and a sufficient test can benefit from this information (will be described later). In case of dynamic workloads where an optimization needs to be executed, some task information needs to be gathered during run-time and isn't known in advance. The description of a task τ_i therefore will be extended by the following dynamic information (cp. section 2.4.2):

- a_i defines the arrival time
- s_i defines the start time
- f_i defines the finishing time
- v_i defines the importance

Considering the optimization, the controller software component decides which task needs to be switched on/off to circumvent any overload situations. The controller hereby utilizes the importance v of each task in comparison to the other tasks and the health of the overall system.

To sum it up, this section has described the model for reasoning within the integration framework. The hereby introduced representations include the ready queues, cores and tasks. As a next step, the following section will describe the integration framework and the usage of this model in more detail.

5.2 Run-Time Task Integration Framework

This section will provide the concept of an integration framework that is responsible for the actual adaptation process. Within this integration framework, the concept of an online admission for dynamic mixed-critical workloads on the same device during run-time will be provided. Starting with a dispatching process that returns a capable core for the given task, section 5.2.1 describes the mapping of a task to a core in more detail. After that, an efficient admission process is used to test if the task fits into the existing ready queue of the core. This will be described in section 5.2.2. For a long-term optimization, a knowledge-based approach was developed within this work which will be explained in section 5.2.3. Finally, the modified task set model needs to be deployed and synchronized from user-space to the kernel-space which will be outlined in section 5.2.4.

5.2.1 Critical-Aware Dispatching of Tasks to Cores

This section describes the dispatching of tasks to cores in more detail. During this process, the attributes of a task are compared with the attributes of the cores (i.e. represented by a *pcore*). A successful matching happens if the task attributes and core attributes are compatible. The dispatching concept was investigated in the work of Nieleck [113]. This section will explain the basic mechanism and ideas behind the matching.

All of the task attributes which are used for the matching process are provided via the static information within the task description. For example, a task must be allocated to a core with equal criticality classification. Furthermore, the scheduling type of a task needs to match the scheduling policy on the targeted core. So, a priority-based task must be allocated to a core with priority-based scheduling. This process is executed for the relevant attributes provided by both task and core. It can be extended for example by the comparison of utilization (cp. bin packing) and other relevant attributes. If a matching was not successful (no suitable core was found), the task will be rejected with a corresponding message. Finding a core which is suitable for the task is based on the following rules:

1. The criticality of a task is compared with the criticality classification of a core:
 $\tau_L = C_\mu$
2. If the task specifies a distinct scheduling strategy, the following check is executed:
 $\tau_S = C_{pol}$ with $pol = EDF, FP$
3. If the task specifies an affinity, the target core with a corresponding id will be used:
 $\tau_A = C_{id}$

With these rules, the concepts in section 4.3.2 for assigning a task to a core can be expressed. However, dependent tasks are assumed to be executed on the same core. In a further step, the selected core is checked for its utilization to clarify if the task in question is able to be integrated onto this core. A capacity-based concept that utilizes an arithmetic approach for checking if the new task fits is used. This is achieved by calculating the utilization or the load of a core and comparing these parameters with the corresponding task parameters. The utilization of a core C_U is represented by the sum of all task utilization within its ready queue $C_U = \sum_{\tau_i} \tau_{U_i}$. The utilization of a task can be provided by $\tau_{U_i} = \frac{\tau_{C_i}}{\tau_{T_i}}$. If a new task with τ_{U_j} needs to be assigned to this core, the following rule will determine if the task should be rejected from execution where θ represents a threshold between $0 < \theta < 1$:

$$C_U + \tau_{U_j} > \theta \Rightarrow \text{reject task}$$

The calculation of the utilization heuristic is based on static information (i.e. τ_{C_i}, τ_{T_i}). This information may not be available in every scenario (i.e. low-critical tasks). An alternative calculation, thus, is based on the actual load of the core which is measured during run-time (i.e. dynamic information). The load can be seen as the amount of time not in the idle task. In this case, the core with the lowest load is taken. If the task

could be assigned to two or more cores, the core with the smallest id will be chosen. The flowchart which is depicted in figure 5.1 describes the overall process in case of a low-critical task [113]:

- An arriving task's attributes are checked against the above mentioned rules
- In case of an available core, the utilization heuristic is used to guarantee that the remaining capacities of this core are sufficient. In a negative case, another potential core will be selected. If there is no other core available, the task will be rejected.
- If more than one core is available and meets the requirements, the core with the lowest load will be selected.
- If more cores with the same load are available, the core with the smallest id will be selected.

If the task defines an affinity A , these checks will be circumvented and the task is assigned to a core with the corresponding id.

The processing steps for dispatching a high-critical task to the system are similar to the outlined flow of integrating a low-critical task. However, due to the fact that a high-critical task requires a highly precise analysis, there is a higher demand of testing. In case of allocating a high-critical task, a complete scheduling analysis could be performed every time a new high-critical task arrives at the system. This approach, however, is inefficient in the context of an embedded system. Another testing approach is therefore selected which can be found in section 5.2.2. Moreover, it will be assumed that a high-critical task need be executed under all conditions. Thus, low-critical tasks can be prevented to allow the execution of a high-critical task. The corresponding steps for realizing such a dispatching behavior are as follows [113]:

- An arriving task's attributes are checked against the above mentioned rules
- If no core is available that meets the requirements, it is checked to determine if there are any unused cores
- If one core is available that meets the requirements, a scheduling analysis has to be performed on that core. If the scheduling analysis is negative, i.e. the task can not be scheduled, the system is checked for unused cores
- If more than one core is available that meets the requirements, the utilization heuristic function is used to determine in which order the cores should undergo a scheduling analysis. The first core that is analyzed positive is allocated the task. For the case that no core turns out to be positive, two possible approaches could be used to try to allocate the task to a core of the system:
 - Immediately try to optimize the scheduling in a way that the task can be scheduled.

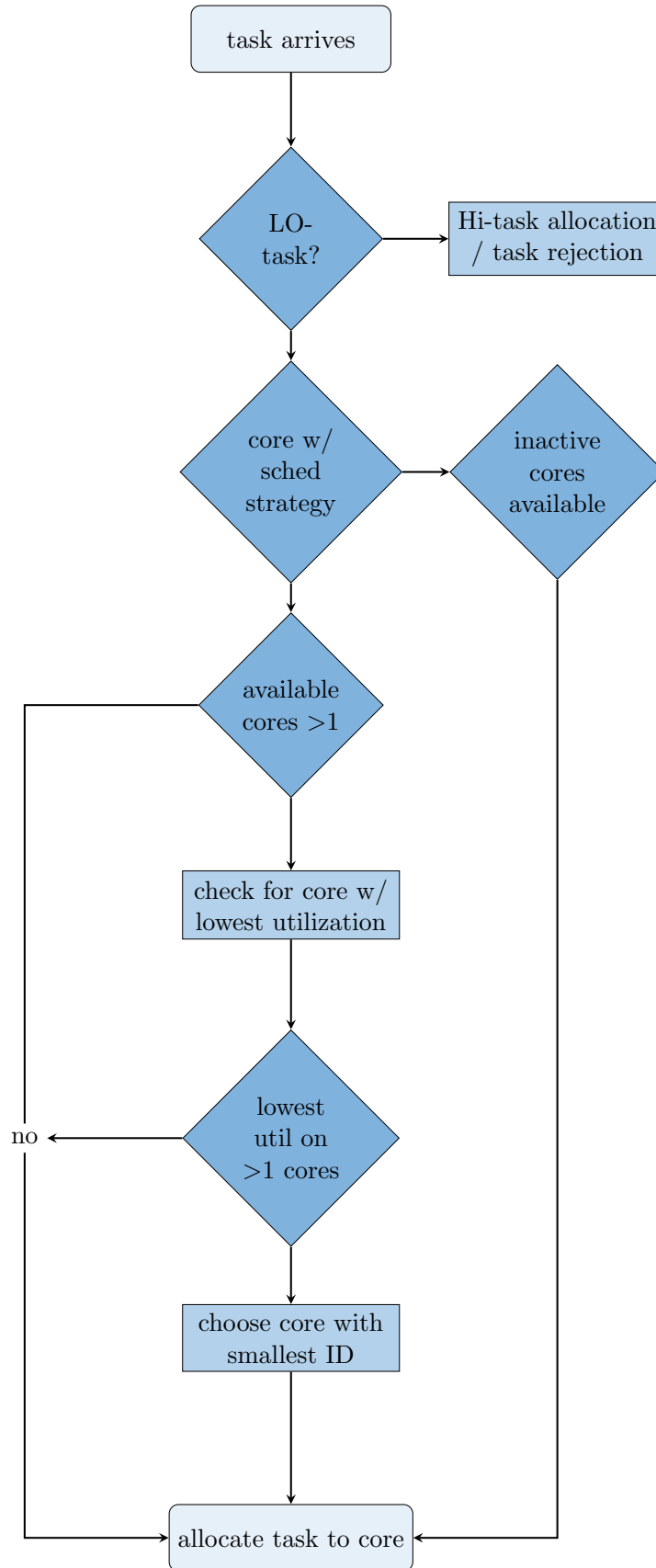


Figure 5.1: Flowchart describing the allocation of newly arriving LO-tasks to a respective core [113]

- Like with the other cases, first check if there is any unused core that can be used. If this is not the case, then optimization could be tried.

The dispatching of a high-critical task is more difficult than the dispatching of a low-critical task. Beside finding an empty core, another solution could be the usage of a propagated low-criticality core. In this case, the low-criticality tasks will be skipped and the core with its corresponding scheduling strategy could be initialized as high-critical. Although this solution is feasible with the provided concept, it disregards the importance of low-critical tasks for the overall system behavior.

In conclusion, this section has described the first part of the integration process in form of a run-time dispatching mechanism. The allocation results in a potential core on which a given task could be allocated. As a next step the concepts for testing the integration of a task into a ready queue of this core will be described.

5.2.2 Short-Term Online Admission Test

The designed admission test is departed in two categories according to the criticality of a task. This section will describe the first category, where a short-term online admission test is used to check if a high-critical task can be integrated in a given ready queue. This approach can be seen as a reactive adaptation. The second testing category, optimizing, will be described in the next section 5.2.3 where this approach can be seen as a proactive adaptation. Due to the context of an embedded system, the used algorithms in both cases need to be resource efficient. As stated before in section 5.2.1, a high-critical task requires an exact analysis in any case. In the best case, a complete schedulability analysis could be used. However, in the context of an embedded system this approach is not practical. For this reason, this section will describe two other concepts. Starting with an online admission approach which utilizes an off-line calculated configuration table, the remaining part of this section will describe an online admission approach which generates such a new configuration during run-time.

Design-Time Configuration and Online Admission

The first approach is using an offline calculated configuration of all possible task combinations which is stored within the system and serving as a knowledge database in case of an admission. Each configuration hereby consists of a fixed set of tasks. Each task gets its priority assigned during the configuration phase. This concept was investigated via a simulation-based approach by the work of Multani [108]. The generation of such configurations is based on a synthesis process which is depicted in figure 5.2 as a flow chart diagram.

The iterative synthesis process for the dynamic reconfiguration *DR-Offline* itself consists of three sub-processes. One process is responsible for the generation of all possible task set configurations. Within this step, τ_L and τ_P will be set. In a second stage, a partitioning process obtains the required dispatch table for different cores using a bin-packing heuristic *BP-FF*. In case of a valid partition (i.e. status feasible), a final task-priority validation process obtains the best possible priorities for the partitioned

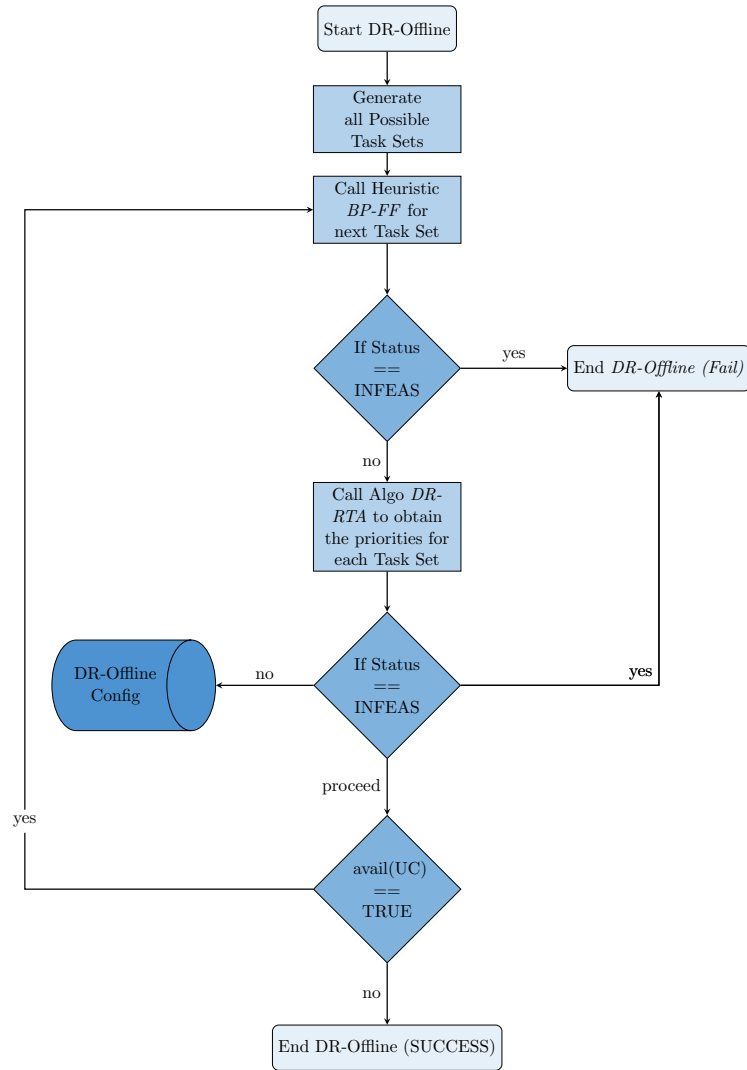


Figure 5.2: Overall Synthesis Process [108]

task-sets. Partitioned task-sets are validated by using a response time analysis *DR-RTA* for each task-set which is allocated to a core. As a result the synthesis process produces highly feasible solutions in form of dispatch tables.

During run-time, adding a new task triggers the admission control which refers to the stored configuration *DR-Offline Config* for selecting the corresponding task set. After that, the ready queue of the matched target core is exchanged with the found configuration. According to the result of the admission process, a positive or negative feedback will be given. The implemented model and simulation framework for testing this concept were using a priority based scheduling strategy [108]. This approach however is independent from the used scheduling strategy.

Table 5.1: Comparison of schedulability analyses for FP [53]

	RTA	HET	Sufficient Test
runtime	+	-	++
power	exact	exact	sufficient
pessimism	++	++	-

Table 5.2: Comparison of schedulability analyses for EDF [53]

	QPA	Algorithm MCF	Approach BHR
runtime	+	++	-
power	exact	sufficient	exact
pessimism	++	-	++

The described approach has the limitation that all tasks sets need to be known in advance. The admission approach, however, need to be able to dynamically add or remove tasks from the ready queues (i.e. supporting an dynamic adaptation). To achieve this, a new approach which provides the calculation of new task sets during run-time will be presented.

Run-Time Configuration and Online Admission

This section will outline a new concept, where ready queues can be configured during run-time (i.e. no offline configuration) on the embedded device itself.

To handle high-critical tasks, the schedulability analysis is replaced by an acceptance test. In accordance to this, the acceptance test consists of a sufficient and an exact test [53]. This acceptance test is capable of taking decisions about newly arriving tasks in an efficient way where the static task information will be used. For example, the used algorithms within this work are using the Deadline τ_{Di} , computation time τ_{Ci} , priority τ_{Pi} and the period τ_{Ti} of a task. The proposed acceptance test can consist of an arbitrary combination of a sufficient test and an exact test. This depends on the underlying scheduling strategies. The outlined concept lies the focus on the usage of priority based scheduling strategies. In the work of Edinger [53], thus several algorithms were identified and validated for a usage in an embedded context. Depending of the schedulability strategy, table 5.1 and table 5.2 summarize the results for a fixed-priority scheduling strategy and an Earliest Deadline First strategy respectively. On the one hand, for a fixed priority preemptive scheduling approach the combination of response-time analysis (RTA) and sufficient test is a good choice. For a core managed by an EDF policy, QPA is a good choice although other algorithms may provide a better run-time complexity.

Integrating the acceptance test in the environment of available system software components leads to the overall process given in figure 5.3. According to the roles of each component (see chapter 4), the task is processed accordingly up to the controller component. After choosing a suitable core via the outlined dispatching mechanism, the task is

checked if it fits into the core's ready queue. This check is done by the available sufficient and exact tests dependent on the required efficiency. If the new task passes one of the tests, it is added to the task-set. Then, in this way, the reconfigured ready queue is exchanged with the current ready queue.

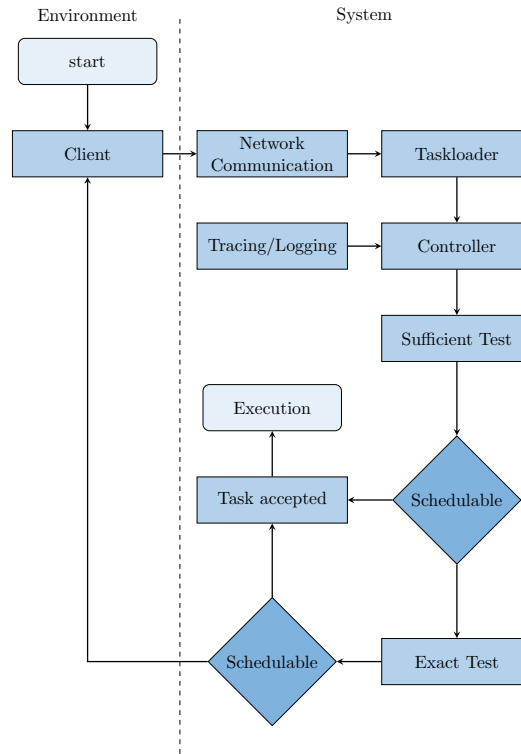


Figure 5.3: Sufficient and Exact Test in the context of other software components [53]

This section has provided two concepts of a short-term admission process for deciding if an arriving task can be integrated within a selected ready queue during run-time. The first concept, hereby, integrates the new task by selecting an offline generated ready queue configuration. The second concept, however, generates this ready queue with the assistance of an acceptance test completely during run-time. Both concepts are able to analyze hard real-time tasks as well as soft real-time tasks. In case of soft real-time tasks, a further admission process, that is able to optimize a given ready queue during run-time, will be presented in the next section.

5.2.3 Long-term Knowledge-based Optimizer

This section will introduce a new concept for the admission of soft real-time tasks during run-time. The goal is to optimize certain ready queues with soft real-time tasks to follow a specified optimization goal. This concept can also be used to continuously integrate a soft-real time task without a previous admission test. In any case, the optimization decision is based on an importance value that is derived from the observed behavior of

each task over time. According to the specified conditions of an optimization goal, the importance value of a task is compared with these conditions. Depending on a positive or negative result, the task is allowed to be executed or not.

If an optimization of a given ready queue is required (i.e. overload situation), a potential optimization decision is bound to a distinct optimization goal. Depending on the desired system behavior after the optimization, it is required that a corresponding optimization goal can be selected. Thus, the proposed concept supports the selection of several optimization goals. These optimization goals are selected according to the criticality aspects and real-time aspects of a given ready queue and its containing tasks. Considering these optimization goals, the optimization approach was tested with certain goals [113], [112], [108]. The focus hereby relies on the fairness, (hardware) utilization and energy consumption whereby each optimization was required to produce a safe state of the system.

Corresponding to a optimization goal, each task holds an importance value τ_{vi} which indicates the behavior of this task under the given goal over time. Beside the (static/dynamic) priority related attributes of a task, this additional attribute is used to classify if a task's behavior correlates with the optimization goal. Hereby, the importance value is the result of a function that calculates the utility of a task depending of the selected goal. This concept aspect was partially investigated in the work of Helminger [70] to decide if a chosen modification of task periods leads to a better schedulability result or not. In contrast, this thesis proposes a concept which generalizes this approach as a solution in user-space rather than a scheduling extension. A decision function which is defined by a distinct optimization goal gets the importance value (i.e. defined by a target function) and checks if this value corresponds with the target requirements of the optimization goal. In either a positive or negative case, the task is allowed or declined to be scheduled.

The basic optimization approach is based around a competing task model [112]. A competing task causes other tasks to violate their timing conditions (e.g. deadline misses). As a countermeasure, the optimizer denies all competing tasks to be scheduled. The optimization process is designed around the following steps:

1. Detect situation where optimization is required (i.e. overload situation)
2. Identification of competing tasks
3. Update scheduling permissions

Situation Detection As a first step the situation in which an optimization is required needs to be detected. In case of a deadline-based scheduling this situation can be recognized by a rising number of deadline misses of the distinct tasks.

Competing Tasks Identification As a second step, the competing tasks need to be identified. Considering that the tasks are scheduled via the EDF scheduling algorithm, a task with a deadline most shortly before the deadline of the considered task (i.e. missing its deadline) can be seen as its competitor.

Scheduling Permissions Update To avoid future executions of a task having a deadline miss due to one of the competing tasks, the optimizer has to reduce the set of tasks which are scheduled by the scheduling algorithm. Due to the reduced set, the remaining tasks may finish their executions before reaching their deadlines.

Using this concept, two approaches for optimizing ready queues according to a fairness goal and an utilization goal were developed in the work of Niedermeier [112]. In case of a fairness optimization, the importance value of a task represents the ratio between the number of missed deadlines in comparison to the number of successfully runs of this task. In case of a deadline miss, the importance value is increased. On the other hand, the importance value is decreased if a task run was successful. As a result, a task with many deadline misses gets a high importance value. The optimizer compares the importance values of all tasks within a ready queue and allows the task with the highest importance values to be scheduled. Thus the selected tasks are able to reduce their number of deadline misses. The fairness hereby is that deadline misses are equally distributed over the task set. Considering the optimization according to the utilization goal, the utilization of a task is assigned to its importance value. Since the utilization value indicates how much processing time the task might require for its execution and the goal is to schedule as many tasks as possible that require a lot of processing time, the optimizer selects the tasks with a high importance value.

This section has described a long-term admission process which can be used to optimize a given ready queue during run-time. The next section will outline a concept for synchronizing the newly generated ready queues from user space to kernel space.

5.2.4 Synchronizing User Space with Kernel Space

This section covers the synchronization mechanisms in more detail. The basic idea is to execute the synchronization exclusively within the operating system. It is important to clarify when the synchronization does happen. In other words, to identify several designated system states where a switch to a new system state does not cause any hazards. For the synchronization itself, there are a common set of synchronization primitives which need to be investigated. Similar to the monitoring component, the synchronizer is heavily dependent from the used system design. Only common synchronization aspects are described in this section where implementation related aspects are covered in the next chapter.

Considering the proposed design, within the user-space a set of new ready-queues are generated which are consumed by the underlying kernel-space. In this case, a new system configuration is deployed into kernel-space. But, arbitrary switching is not allowed due to safety restrictions which implies a robust and reliable system despite of adaptation. Additionally, it can't be assumed that the system is in a designated state (e.g. idle) at the time of switching request. Therefore, some kind of synchronization between the user-space and the kernel-space needs to be established.

Possible synchronization primitives for the usage in embedded systems, as part of a user-space realization, were analyzed in the work of Haecker [66]. The primitives

can be categorized in lock-based and lock-free mechanisms. In conclusion, table 5.3 summarizes the compared synchronization primitives and shows that a sequential-lock based mechanism provides a potential solution.

Name	Deadlocks	Read-Speed	Write-Speed	Security	Overhead
Semaphore/Mutex	1	1	1	5	1
Read-/Write-Lock	3	3	1	5	3
Sequential-Lock	5	4	5	5	3
Read-Copy-Update	5	5	5	3	3

Table 5.3: Comparison of lock-based and lock-free mechanism [66]

An actual realization and deeper investigation of the found synchronization mechanism was done in the work of Chandrasekhara [33]. The evaluation result can be found in figure 5.4 which confirms that a lock-based mechanism provides a better solution for an embedded system in comparison to a lock-free mechanism like Read-Copy-Update (RCU).

Criteria	Mutex	RCU	STM
Implementation	+	-	++
Read-speed	-	++	+
Write-speed	-	+	+
Deadlocks	-	++	+
Overhead	+	+	-
Security/Consistency	++	-	+

Table 5.4: Comparison between Mutex, RCU and STM [33]

Regardless of the used synchronization primitive, another important aspect is the point in time when a synchronization needs to take place. In other words, when the created task-set is deployed into the underlying kernel. This point depends on several conditions: the used scheduling and the state of distinct ready queues. In this scenario the focus lies on static priority preemptive approaches as well as dynamic approaches. Three conceptual approaches for possible concepts were investigated in the work of Chandrasekhara [33]:

- **Empty ready-queue:** If the run queue is empty, exchanging the list is easy since no threads exist in the ready queue. In this case, it is assumed that the exchange of the entire ready queue list with a new list can be done. A CPU lock is used to disable the thread preemption and the new ready queue will be deployed.
- **Static point-in-time:** Regarding a static point-in-time switching policy within a filled queue, a selected thread (e.g. idle) can be used to determine the end of an execution cycle. The switch takes place after the end of the execution cycle. A controller component decides which thread should be allowed to complete its execution before exchanging the ready queue.

- Variable point-in-time: The synchronization approach can be realized by using a lock mechanism that supports a variable point-in-time for switching (i.e. dynamic synchronization point). Potential mechanisms are Read-Copy-Update (RCU) or Software-Transactional-Memory (STM). But, these mechanisms come with a cost of implementation complexity and thus overhead.

To sum it up, this section has outlined a synchronization concept that is required for the exchange of task sets from user-space to the underlying kernel-space. For that reason, several algorithms are evaluated and suitable candidates were identified. Further, several ready queue states were investigated for the concept.

5.3 Combine the Framework Concept with the Designed Architecture

Within this section, the presented run-time integration framework concept will be combined with the designed consolidated architecture from chapter 4. The focus hereby will rely on the software components tracing/logging, controller and synchronizer which are mainly realizing the already presented concepts. For an overview, figure 5.4 depicts these software components as well as their data flows. Starting with section 5.3.2, this section will briefly outline the usage of co-existent scheduling strategies. After that, section 5.3.2 will provide details about the tracing/logging software component including a monitoring process for a microkernel-based operating system. Section 5.3.3 will give an overview about the controller component. This section will close with details about the synchronizer component (see section 5.3.4).

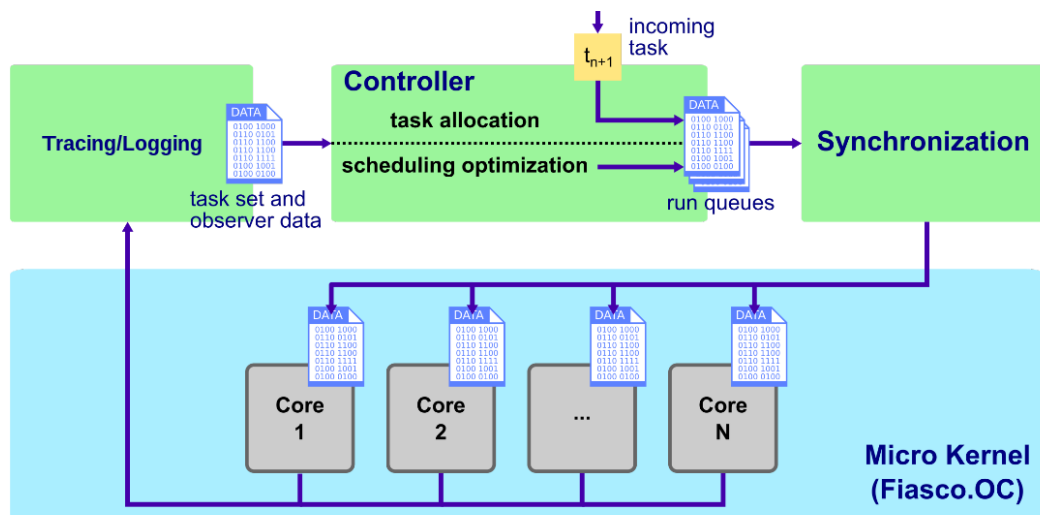


Figure 5.4: Abstract system overview with tracing/logging, controller, and synchronization [113]

5.3.1 Co-Existent Scheduling Strategies

Partitioned scheduling scheme was selected because of a better predictability and smaller overhead by using a partitioned scheduling approach. Each scheduler can be treated as one single core scheduler within a multi-core based setting. For that reason, Multani investigates several scheduling scheme alternatives to find a feasible combination of admission and enforcement [108]. As a result, the concept of co-existent scheduling strategies will be realized through the usage of an (dynamic) earliest-deadline first and a (static) fixed-priority preemptive scheduling strategy within the kernel.

5.3.2 Tracing/Logging Software Component

The tracing/logging software component is supporting the decision process within the integration framework by delivering information about the current system state to the software components controller and synchronizer. Obviously, the mechanism to get relevant information about the current scheduling state (i.e. ready queues) out of the underlying system highly depends on the provided kernel interface. Thus, details for a concrete implementation of such a mechanism will be described in the following chapter 6. This section will provide a conceptual overview about the monitoring mechanism within the tracing/logging software component.

Due to the fact that the decision process within the integration framework is executed online during run-time, it is required that the monitoring mechanism also provides its information during run-time. Moreover, the monitoring mechanism needs to be able to deliver its information in a real-time setting because the proposed management approach is targeting this kind of systems. With these requirements, several tracing mechanisms were investigated and their suitability was tested for the outlined monitoring approach [119] within the designed architecture. As a result, an existing tracing framework could be used as a starting point for the monitoring mechanism. The tracing framework, however, has provided the basic mechanism which were extended by the required interfaces and information to build a sound monitoring solution.

The outlined task model in section 5.1.2 (i.e. application description) contains the static and dynamic parameters of a task and its corresponding thread. The monitoring method delivers the dynamic information to populate dynamic parameters of the task model. The dynamic information hereby summarizes the efforts of several works [92, 64, 113] to find a consistent representation of required monitoring information for the outlined adaptation workflows. The monitoring process is working on a representation of exactly this task model that is implemented as a stand-alone monitoring object (see chapter 6). Thus, a consistent management of information during the integration process could be achieved.

This section has outlined the conceptual contents of the tracing/logging software component. A real-time monitoring mechanism that is executed during run-time is the central part of the concept. An existing tracing mechanism which supports the run-time and real-time requirements was extended to a sound monitoring solution for the proposed integration framework. The next section will outline another software component where

a great majority of the decision process will take place, namely the controller software component.

5.3.3 Controller Software Component

The controller software component is realizing the following processing steps of the run-time integration framework:

- critical aware dispatching of tasks to cores
- short-term online admission testing if a task fits into the system
- long-term knowledge-based optimizing of existing ready queues

This section will introduce the concrete analysis strategies and mechanism that are selected for the realization of the outlined concept within the controller component. A concrete implementation of the component, however, will be shown in the following chapter 6.

An abstract overview of the controller process (according to Nieleck [113]) with its parts can be seen in figure 5.5 where the “input to the controller is a set of tasks that are executed on the system and the monitored system state” [113]. The controller is able to modify the blue circled aspects of the system, namely allocation, tasks, cores and scheduling strategies. These modifications are constrained by one or more attributes (red box) e.g. overload. As a result, the controller constructs several task sets (run queues) which are bound to distinct cores. The tasks within the task set are executed on the corresponding core. The individual parts of the controller are explained in the following.

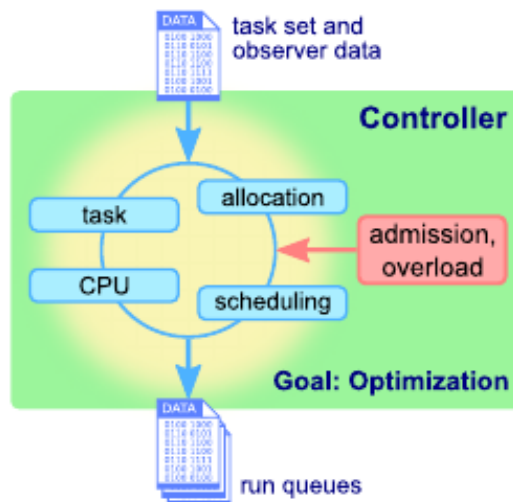


Figure 5.5: Abstract overview of the optimization process of the controller software component [113]

5.3 Combine the Framework Concept with the Designed Architecture

For a critical aware dispatching of tasks to cores, the controller stores an abstract representation of the cores and their corresponding ready queues (cp. concept in section 5.1.1). The dispatching algorithm itself is designed according to the concept that is presented in section 5.2.1. An efficient matching process serves a central part of the designed dispatching algorithm. This matching process basically compares the attributes of a task description with the attributes of the existing cores. A feasibility test of this matching process is given in the work of Nieleck [113] via a simulation-based testing approach.

Considering the concept of a short-term online admission test during run-time that is provided in section 5.2.2, a concrete combination of an exact test and a sufficient test is required. As a potential combination for a fixed-priority scheduling, Edinger [53] has proposed the combination of a response time analysis and Bini's sufficient test. The algorithms are using the user-space representation of the ready queues like proposed in the concept (see section 5.1.1). For the implementation of the algorithm (see chapter 6), it is required that the ready queues are sorted. An incoming task will be checked if it fits into the system or not according to the admission concept in section 5.2.2. Similar to the critical aware dispatching of a task, these algorithms need to be efficient. The concept was finally tested in the work of Edinger [53] and it could be demonstrated that this combination shows it is capable as an online test during run-time.

For the concept of a long-term optimization of existing ready queues, a concrete optimizing process for the earliest deadline first is realized within the controller software component. Following the concept in section 5.2.3 an algorithm which is able to avoid overload situations was integrated. As outlined in the concept, a fairness and utilization optimization can be selected as target function. Additionally, this optimization goal can be changed during run-time via a dedicated control interface. The optimizer process, hereby, is realized for the optimization of low-critical cores. An application of high-critical task sets is feasible but were not investigated so far. For realizing the concept of concurrent tasks, the controller additionally stores a list with information of concurrent tasks, deadline misses and execution states. With this information the optimizer is able to use the history of a task for its decision. In her work, Niedermeier [112] shows a polynomial timing behavior via a static analysis of the algorithm. However, a later run-time evaluation could demonstrate that the optimizing algorithm works like expected.

This section has provided the concrete combination of the provided concepts for dispatching and admission with the controller software component. All concepts could be successfully integrated and tested as part of the controller component. The next section will provide more details about the synchronizer software component.

5.3.4 Synchronizer Software Component

This section will describe the synchronizer software component which realizes the concept in section 5.2.4 for synchronizing the user-space and the kernel-space.

The synchronizer component is completely realized in the user-space with no direct access to the kernel ready queue objects. The synchronizer uses the same user-space ready queue representation like the tracing/logging and the controller software component (see

concept 5.1.1). Thus, there is no separate kernel object required which corresponds to the idea of a small trusted computing base. An existing scheduler kernel object, however, needs to be extended by a deployment system call interface. A first realization and test of a user-space synchronizer software component was done in the work of Chandrasekhara [33]. The synchronization was realized by a per-cpu variable for locking the CPU and disabling the interrupts to update the existent ready queue. Corresponding to the process of finding an adequate point in time for synchronizing user-space and kernel-space, a smart-sync labeled method was realized. This method provides a consistent synchronization check that consists of testing if a ready queue is empty or a scheduler is idle (i.e. static point in time for synchronization). The synchronizer component is able to interchange single threads or complete ready queues.

This final section has described the synchronization software component and the combination with the former outlined concept of synchronizing the user-space and kernel-space.

Conclusion

This section has provided the conceptual foundations about the flexible task management which supports the self-adaptation of a system from user-space. Therefore, a user-space model for representing the system state as well as a single application was introduced. After that, the concept of a run-time integration framework was outlined which divides the integration process into distinct steps. The focus hereby has relied on the dispatching, admission and synchronization of tasks. The chapter has closed with transferring the developed concepts to the designed software components. A possible implementation will be shown in the next chapter 6.

6 Extension of a Microkernel-Based Operating System

In this chapter, relevant implementation steps for realizing the architecture as well as the outlined concept will be provided. This chapter thus corresponds to the implementation contributions (contribution category number 4). Starting with basic concepts of the used microkernel-based operating system, section 6.1 outlines by Fiasco.OC and Genode provided concepts. The focus in the consecutive section 6.2 relies on the required extensions within the micro-kernel for supporting the proposed concept. Followed by section 6.3, the concrete implementation of relevant operating system components for the flexible task management will be outlined. This chapter will close with current limitations of the implemented extensions (see section 6.4).

6.1 Genode and Fiasco.OC Basic Concepts

The basic concepts of the L4 Fiasco.OC microkernel (in the following abbreviated as Fiasco.OC) and the Genode OS Framework (in the following abbreviated as Genode) will be covered in this section. At a first step, the overall picture of the interplay between Fiasco.OC and Genode will be outlined. Moreover the distinct roles and features of each software part will be explained. A deeper investigation of relevant information about Fiasco.OC addresses the basic architecture, scheduling context, capabilities concept and system call interface. Genode complements these mechanisms by a consistent software stack where the basic architecture and the main concept of client/server infrastructure are subject of the section which will follow afterwards.

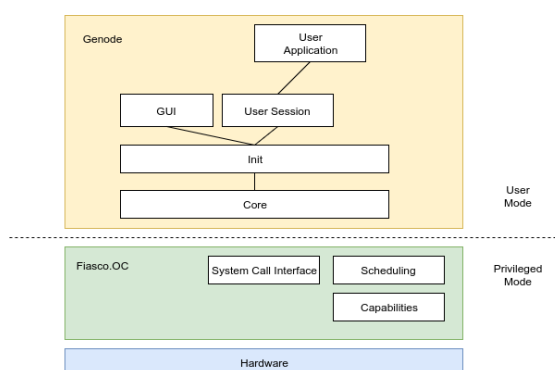


Figure 6.1: Overall Overview about Fiasco.OC and Genode

The simplified architecture depicted in figure 6.1 shows the overall relation between Fiasco.OC and Genode. The microkernel, Fiasco.OC, thereby is executed in privileged mode and provides a mostly policy free (i.e. except scheduling) abstraction to the underlying hardware. Relevant aspects for possible extensions in the context of this work are the system call interface, the scheduling and the capability system. Concrete policies for realizing the distinct services of an operating system (e.g. device drivers, protocol stacks etc.) are left to Genode which is executed in user-space. It follows a recursive system architecture based on a tree-like parent/child concept. The root of this tree is *core* which has access to the raw physical resources such as memory, CPUs etc. In combination with *init* (started by core), core provides the basic policies for higher-level components (e.g. GUI session and Applications). Init instantiates subsequent child-node components. Deeper insights of Fiasco.OC and Genode considering the provided aspects will be presented in subsection 6.1.1 and 6.1.2.

6.1.1 L4 Fiasco.OC Microkernel

This section describes the L4 Fiasco.OC microkernel in more detail. As a basic idea, Fiasco.OC strictly follows the definition of a microkernel in the sense of separating policies from mechanisms. Mechanisms are therefore provided by the microkernel where applicable (i.e. exception is scheduling) and are complemented by a concrete policy within the user-space. Thus, Fiasco.OC realizes basis functionality like processing (i.e. scheduling), resource management, and communication. Fiasco.OC therefore provides several kernel objects which realize the main functionalities of the kernel (i.e. communication, memory management and processing). The following provides an overview of all kernel objects with a short explanation of their distinct role.

- **IPC Gate:** Used for communication.
- **Task:** Provides the memory for the threads and encapsulate the threads in a protection domain.
- **Thread:** As a central element a thread represents the driver class for most kernel functionality.
- **IRQs:** Handler for interrupts raised by the underlying hardware. Hardware interrupts. This class encapsulates hardware IRQs. Also, it provides a registry that ensures that only one receiver can sign up to receive interrupt IPC messages.
- **Scheduler:** Provides access to all processing related information and actions (i.e. ready queues)
- **Factory:** Special kernel object which is used to create all other kernel objects via a defined interface. An exception hereby is the scheduler object which cannot be created over this interface.

The processing management of Fiasco.OC is based on tasks, threads and a scheduling context. In this context, tasks are defined as protection domains which realize a virtual address space (i.e. separation of tasks). Within each task one or more threads are executed. Each thread owns a kernel-level thread control block (ktcb) consisting of registers, instruction pointer and stack pointer as well as an user-level thread control block (utcb). The utcb is meant as communication buffer for inter-process communication between user-space and kernel-space. Figure 6.2 depicts the basic relationships between a scheduling context, a thread and a task in a simplified form. Each task references a thread which in turn is derived from the context class which stores a reference on a *scheduling context* instance. Different scheduling parameters which are encapsulated with a scheduling context (e.g. `start_time`) are assigned to each thread. Fiasco.OC uses these information to determine the execution order. The scheduler is working on ready queues which are also encapsulated in a scheduling context. Each instance of a ready queue is realized in a distinct class and may be used in a coexistent way. The standard scheduling strategy is a round robin algorithm which gets its thread information from the corresponding scheduling context (e.g. priority).

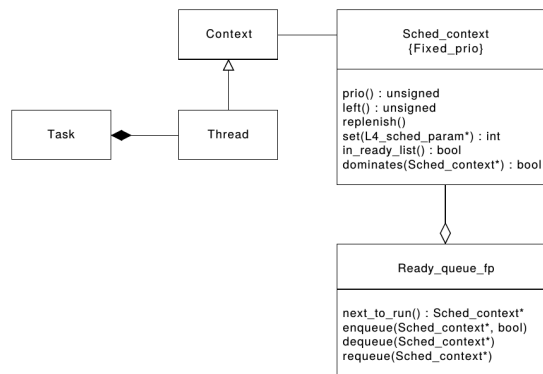


Figure 6.2: A ready-queue with threads containing their scheduling parameter is assigned to each core. A scheduler populates this ready queue. [68]

A key concept of Fiasco.OC is the usage of capabilities for the communication between kernel objects and user-space components. The basic idea is to circumvent security flaws due to the avoidance of global object identification. In contrast, each task gets a list of capabilities which is used to identify other objects with a local identifier. Tasks are able to communicate with other tasks if they have the corresponding capability. For the communication with another object, the task binds the communication on a capability. Moreover, each capability-based access is routed through the kernel which checks the permission in a first step. If a task is permitted (client), the kernel grants the access to the communication capability of the receiver task (service provider) by simultaneously resolving the receiver capability of the receiver task.

Another key aspect of micro-kernels and Fiasco.OC in special, is the Inter Process Communication (IPC) as a central part of the communication between kernel objects and between user-space components. The synchronous IPC mechanism (i.e. used by Fiasco.OC) is similar to function calls where the receiver signals the possibility to receive data. If a sender needs to send data to the receiver, the former is blocked by the kernel and the receiver is able to read the data directly from the storage of the sender. But also for the communication between kernel space and user-space, the ipc mechanism is used in form of system calls. Due to the fact that the IPC-calls are implemented deeply within Fiasco.OC (i.e. assembly), it can be guaranteed that every communication is routed through the kernel and thus allows the observation of the communication and potential access violations (i.e. capabilities).

For the communication during an IPC call, the information is stored in distinct registers and buffers. Buffer registers are used to transfer the information indirectly to the receiver via shared memory. A direct copy is sent to the receiver by using distinct message registers (i.e. within utcb).

Services which are provided by the kernel are exclusively callable by IPC requests. The IPC framework however doesn't define a distinct protocol, thus for each service a protocol need to be implemented. As an example, the scheduling protocol describes the direct communication process between user-space components and kernel scheduler objects. Basically, the scheduler interface can be used if the required capabilities are given. Table 6.1 describes the individual scheduling operations provided by the communication interface in more detail.

L4 PROTO SCHEDULER	
Operation	Description
L4.SCHEDULER_INFO_OP	Get general information about the scheduler (i.e. number of cores)
L4.SCHEDULER_RUN_THREAD_OP	Thread executed
L4.SCHEDULER_IDLE_TIME_OP	Get information about computation time

Table 6.1: Scheduler Operation

This section has described the already available mechanism in Fiasco.OC for processing, memory management and communication. In a next step, section 6.1.2 will cover the available mechanism in Genode.

6.1.2 Genode Operating System Framework

This section will covers the Genode Operating System Framework in more detail. Especially the unique concepts considering the architecture, communication and right management will be described. A basic design decision of Genode is to explicitly declare the resource usage where in contrast to monolithic operating systems, Genode does not try to abstract from physical resources and does not provide an application with the impression that it has access to (almost) unlimited resources [56]. Genode is a kernel-agnostic solution where micro-kernel system calls are abstracted in a way that it resembles the general separation model from policy and method as closely as possible. Stemming from the kernel capability management, Genode's architecture always follows a tree-like

pattern where the right management is applicable in a recursive way from the root to the leafs. The overall communication is handled by remote procedure calls which concrete the more abstract inter process communication framework found in the Fiasco.OC micro-kernel. At first, more insights about Genode's capability system as well as its component architecture is given. The section closes with the communication infrastructure of Genode.

Similar to Fiasco.OC, Genode binds capabilities to resources. Resources (e.g. cpu session) are represented by RPC (Remote Procedure Call) objects which can be called if the proper rights are given. RPC objects provide a clearly defined interface through a component (i.e. process) may leave its own sandbox (i.e. protection domain) and interact with the remote object. In comparison with the basic kernel IPC framework, accessing a RPC object with a capability results in a access test within the kernel where the already introduced capability list is used to check if the access is allowed. Genode translates kernel capabilities in an abstract representation under the usage of kernel-specific implementations (i.e. each kernel gets its own capability implementation in user-space). Moreover, capabilities can be transferred (similar to Fiasco.OC) whereby a resource can be accessed by a process which holds the right capability or by directly requesting the resource capability. Requesting capabilities is not limited to resources held by other components. Similarly, if a component needs to start an additional thread, it uses its CPU session capability to ask its dedicated CPU session object at Genode's *core* component to create and set-up a thread for the client component.

Genode's resource management is based on a parent-child relationship between the components. A parent is responsible to provide the necessary resources during the creation process of its children where it donates a part of its own resource budget. Regardless of how many (sub-) children a parent creates, the total amount of resources for the whole branch is limited by the parents initial budget (initial configured during run time).

The communication within Genode is based on a server-client paradigm with remote procedure calls. When a component needs to perform an operation, which is not internally provided by it, it needs to ask a another server component to provide it with this service. A component however could be both a server (i.e. service provider) and a client (i.e. service consumer). Server provided services are announced to their parent during the creation of the component by passing the service name and a special root capability to the parent.

Communication between Components

The basic inter-process communication mechanism is the synchronous remote procedure call. The RPC interface is designed similar to the ipc realized by Fiasco.OC where it closely resembles the semantics of a function call, where the control and thus the data is transferred from the caller to the called component. Genodes RPC mechanism is built on top of the kernel's IPC mechanism which explains the similarity to Fiasco.OC.

For the RPC communication there are three main parties involved, namely server, client and the kernel. The situation is the same as in similar user-space implementa-

tions like L4Re where neither the client nor the server are aware of the identity of the corresponding communication partner. The access control (i.e. identity check) comes in form of capabilities which are validated within the kernel. The integrity of this capability based approach is guaranteed by the exclusive management of kernel objects and corresponding protection domains each associated with a capability inside the kernel. Furthermore, Genode introduces a strategy where a parent is in charge to decide if a child component is going to receive certain capabilities [64].

If a component needs to communicate with other components in the Genode system, the component establishes a connection to other components to use their services. All components are also responsible for routing service requests to their corresponding services. After the component has established the connection it is able to directly communicate with the server (i.e. via a capability). There exists several session types within the Genode system, but the CPU session associates the component with a kernel process to be scheduled by the scheduler. Preceding child creation, the parent is required to create a CPU connection for the child, specifying session name and its scheduling priority, as will be described later.

For the communication itself, there are several implementation classes responsible which are depicted in figure 6.3 and are described by Guba [64] as follows:

- **Session** The Session class is the central interface between client and server.
- **Client** The Client class holds the capability to use an RPC Session interface.
- **Connection** Interacting directly with the Client class is still rather complex, as the Client needs to be initialized using session-construction arguments, connected, and closed again. In order to simplify this process, the Connection class encapsulates a Client and provides a single interface for a component to use.
- **Session** The Session Component class is the actual RPC object that is to be Component called. It features implementations of the functions defined in the Session interface.
- **Root** The Root Component class, similar to the Connection class on the client Component side, encapsulates the usage of a Session Component. It announces the service by name and with it the capability to create RPC sessions to the parent.

Beside RPC, there are two other mechanisms for inter-component communication within Genode - signals and shared memory. Signals are useful when a component needs to get notified about events in an asynchronous way (i.e. contrary to RPC, the control is not delegated to another component). For sharing huge amounts of data, the usage of shared memory for this case is foreseen in Genode. One component has to provide the shared memory region (e.g. dataspace) out of its own RAM quota where one or more other client components could attach this region to their virtual address space (i.e. capabilities are granted) and are able to effectively access the same physical address space range [56].

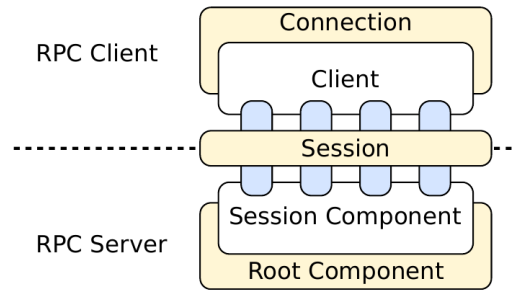


Figure 6.3: Genode inter-process communication [64]

Communication between Kernel-Space and User-Space

As an starting point, if a given user-space software component wants to get related timing information (e.g. period) out of the kernel. The former described user thread control block is thereby used as buffer to interchange relevant information between kernel and user-space. The utcb will be supplemented by message registers (mr) which contain additional (meta-)information about e.g. the target operation (e.g. *get_information* about threads. The actual kernel system call is encoded in one single `ipc_call` which makes a routing according to the target function within the kernel necessary.

The collected kernel information can be placed within a user-space *trace buffer* which is provided by Genodes tracing framework. “Tracing enables a monitor process to provide some of its own RAM quota to log information about other processes” [64]. Genode’s *TRACE* service is required to access the buffer and trace events. The provided service does not exclusively provide information about traced processes rather than any process currently running within Genode.

To sum it up, this section has described the basic concepts of Fiasco.OC as well as Genode. After identifying the important aspects and interfaces, the required extensions for implementing a flexible task management on top of Genode will be described. Starting with the changes in Fiasco.OC the section 6.2 describes the modifications in greater detail. After that it follows the implementation of relevant user-space components within Genode.

6.2 Extension of the Fiasco.OC Microkernel

This section describes the necessary and realized changes within the implementation of Fiasco.OC (see figure 6.4). First, the kernel is extended by an additional scheduling policy to provide the at least two different scheduling policies which can be executed in a coexistent way (see section 6.2.1). For an extended monitoring of the kernel itself, additional information about timing aspects of individual threads needs to be enhanced (see section 6.2.2). This information is later used within the user-space to e.g. get system information. But also the other direction, from user-space to kernel-space, exists (see section 6.2.3). Starting in the user-space, new system states (i.e. queues) need to be

6 Extension of a Microkernel-Based Operating System

transferred into the kernel. Moreover, there is other information for a later analysis of the running system which is collected inside the kernel. An overview about the relevant information as well as their implementation will be given.

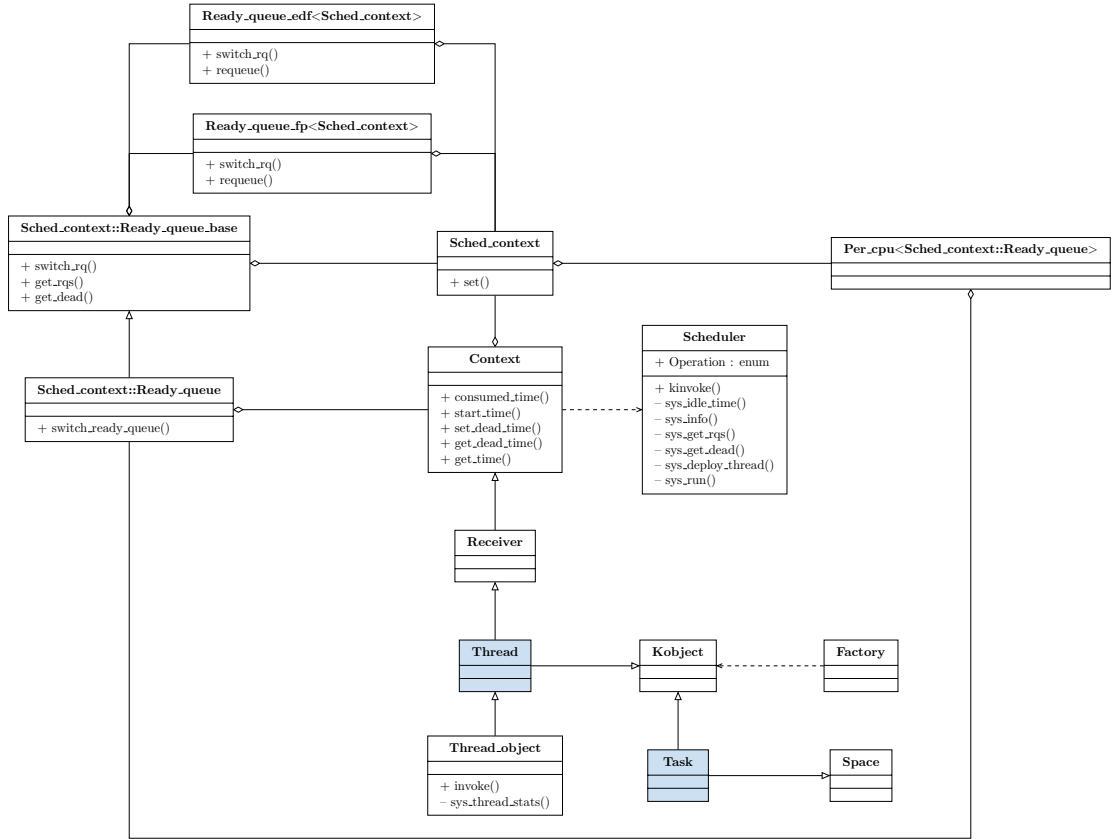


Figure 6.4: Collaboration Diagram for Fiasco.OC Microkernel

6.2.1 Adding additional Scheduling Policies

To provide a coexistent scheduling scenario, Fiasco.OC need to be enhanced by an additional scheduling policy. In the concrete case, the earliest deadline first strategy is realized within the kernel (i.e. `Ready_queue.edf`). Starting with a main challenge for the potential extension, it follows a possible solution where also limitations of this solution are highlighted.

For describing the main problem for a potential solution it is noteworthy that Fiasco.OC is designed according to a partitioned scheduling approach. Each core in the system gets its own ready queue which is managed by the corresponding scheduling strategy. Ready queue and strategy are encapsulated in a scheduling context. But, for the whole system only one scheduling context is allowed. This means all cores are managed by the same scheduling strategy like fixed priority preemptive. For the execution of

coexistent scheduling strategies this approach is obviously a limiting factor. The possible solution approaches are discussed in the following.

In a first solution [52], a dedicated EDF scheduling strategy was intended to be executed beside a fixed priority strategy (i.e. pure coexistent). For a proper realization of this approach, the complete building process of Fiasco.OC needs to be extended because of the circumstance when a scheduling strategy is assigned to the core. This happens during the compilation process of Fiasco.OC where each core gets a dedicated strategy via a *Makefile* based configuration procedure. Adapting the complete building process is a very time consuming challenge. Furthermore, the scheduling strategies are fixed assigned to a core and therefore not interchangeable during run-time. Due to the lack of enough flexibility and the time consuming modifications, an other solution needs to be found.

Another solution which is based on a former idea of [68] allows the application of coexistent scheduling strategies without a time consuming re-factoring of the toolchain. The former distinct strategies of fixed priority and earliest deadline first will be combined. The resulting combined strategy (i.e. scheduling context) consists of two ready queues (one per strategy) which are assigned to one core. Considering special thread parameter, it will be decided in which target queue the thread will be placed in.

Starting in user-space, the creation of threads is handled by Genode's *Platform_thread* class. The *finalize_construction* method as shown in listing A.1 was extended to assign a distinct parameter to each thread which can be used to decide about a thread's affiliated scheduling strategy. There are currently two parameters supported which can be assigned via the modified *l4_sched_param_by_type* method: *Deadline* and *Fixed_prio* indicating that the corresponding scheduling strategy is either earliest deadline first (EDF) or fixed priority preemptive (FP). These settings are propagated to the underlying kernel by running the created thread via the *l4_scheduler_run_thread* system call.

Fiasco.OC provides a high-level interface of system calls which are based on the common IPC kernel interface. In this case, the function *l4_scheduler_run_thread_u* as shown in listing A.2 is used to prepare the required message registers for the resulting *l4_ipc_call*. Within this function the priority or deadline parameter will be set. Additionally, the requested operation code for running a thread (i.e. *L4_SCHEDULER_RUN_THREAD_OP*) as well as the target kernel object (i.e. *L4_PROTO_SCHEDULER*) needs to be set. Within the scheduler kernel object, the IPC call is routed to an internal *sys_run* (see listing A.3) method by using the former requested operation code. At this point, the former assigned attributes for priority and deadline are used to check which scheduling context should be used. Finally, the corresponding scheduling context will be set by calling the set function of the scheduling context as shown in listing A.4. The setting of the adequate context determines the resulting scheduling queue. In this case, the scheduling context consists of two subcontexts each wrapping a dedicated ready queue. As a result the thread will be enqueued in the contexts ready queue which is either *Ready_queue_fp* or *Ready_queue_edf*. Each queue class implements the corresponding strategy.

After describing the extension of Fiasco.OC with an additionally scheduling policy within a combined scheduling context, a next step is to get information out of the kernel.

6.2.2 Extension of Time-related Thread Information

Providing the rich information about individual threads considering their timing behavior requires an extension of current thread structures within the kernel. The missing information which is derived from the presented task model are implemented accordingly. For a later analysis via a response time analysis or other optimizing mechanism, run-time information about threads is required. Where an user-space monitoring component (see section 6.3.2) receives and provides relevant information to other user-space components. This section follows the logical order of the systems call graph for getting relevant information out of the kernel (depicted in figure 6.5).

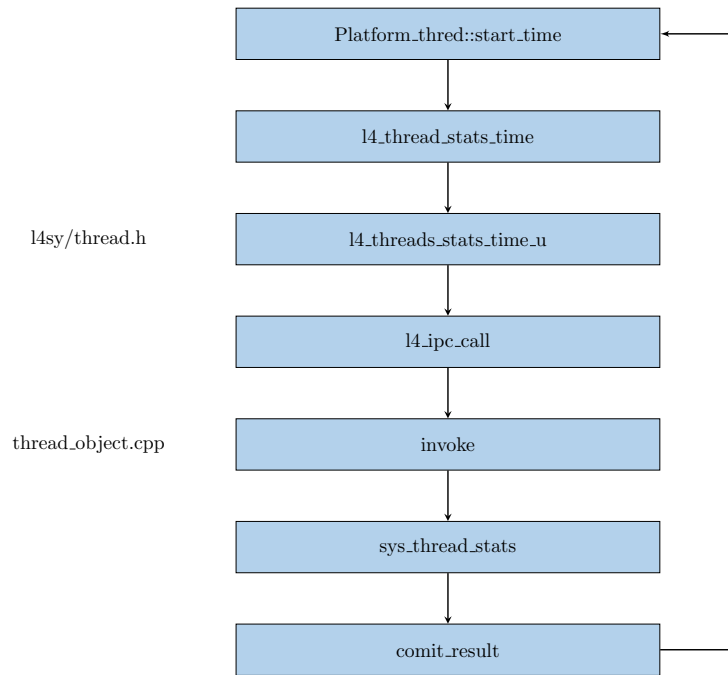


Figure 6.5: Simplified flow of Fiasco.OC and Genode interface

At first, for providing relevant timing information of a thread, Genode’s *Platform_thread* class was extended by demanded methods. Considering the former described task model, the following timing attributes of a thread can be identified: consumed time, period, starting time, finishing time and system time. Each attribute gets its respective method where some of these methods need to get their information directly from the kernel. For example, the *start_time* method collects its information by calling the appropriate kernel *l4_thread_stats_time* function (see listing A.5). As previously stated, each function call prefixed with *l4_* triggers a corresponding internal system call within the high-level kernel interface (i.e. *l4sys*). The internal system call, in this case *l4_thread_stats_time_u* (see listing A.6), takes care of preparing distinct message registers and calling the kernel’s ipc method. Within the registers, the operation code (here: *L4_THREAD_STATS_OP*) and the *Thread* kernel object will be set as usual.

Finally, the kernel's *Thread_object* class is responsible for handling the corresponding IPC call considering the operation code. This decision consists of several steps which involves distinct methods like the *invoke* function (see listing A.7) of the *Thread* class where the corresponding operation code handler will be selected. For each new thread timing attribute which should be provided by the kernel, this method need to be modified. The operation codes are hereby assigned to concrete function calls within the *Thread_object* class. In case of getting the start time of a thread, the corresponding method is *sys_thread_stat* which is shown in listing A.8. This function aggregates diverse thread information mainly from the kernel scheduling context object (*Scheduling_context*) and sends them back to the user-space by calling the *commit_result* method. The scheduling context object of the kernel stores related timing attributes in general. Missing timing attributes were supplemented (i.e. required calculations) as well as corresponding methods which allow the access of the novel attributes. The interface of the scheduling context object therefore were extended by these functions (see listing A.9). As a result, required timing attributes are accessible in a consistent way.

To sum it up, this section has described the flow of timing information from kernel to user-space. In the next step, the other way around will be described to deploy threads into the kernel during run-time.

6.2.3 Extension of Scheduling-Context for Thread Deployment and Information Gathering

This section describes the extension of Fiasco.OC to be capable of deploying new threads into the kernel. The default scheduling context is mostly (but not limited to) focused on fixed-priority scheduling (as its default scheduling policy). A deployment of new tasks or switching of complete task sets (i.e. ready queues) is yet not possible. The extension of the scheduling context is divided in four steps:

- Implementing the actual deploy function
- Creating a reference within the invoke function of the ipc_call interface
- Creating a new capability for the call
- Implementing the actual switching method for single threads or sets of threads

In general, this extension is meant to transfer the information from user-space to kernel-space (in contrast to section 6.2.2). Nevertheless, in the context of this section additional changes considering the scheduling context which allows gathering information about the ready queues per core will be described. Similar to section 6.2.2, a call graph (depicted in figure 6.6) is used to provide an overview about the involved functions (i.e. distinct code changes).

To provide the Genode's *Platform_thread* class the opportunity to deploy a thread to the kernel during run-time, a corresponding function within Fiasco.OC's *l4_sys* interface is required. This function is labeled as *l4_scheduler_deploy_thread* and is shown in listing A.10. The corresponding changes, namely a new operation code as well as an internal

6 Extension of a Microkernel-Based Operating System

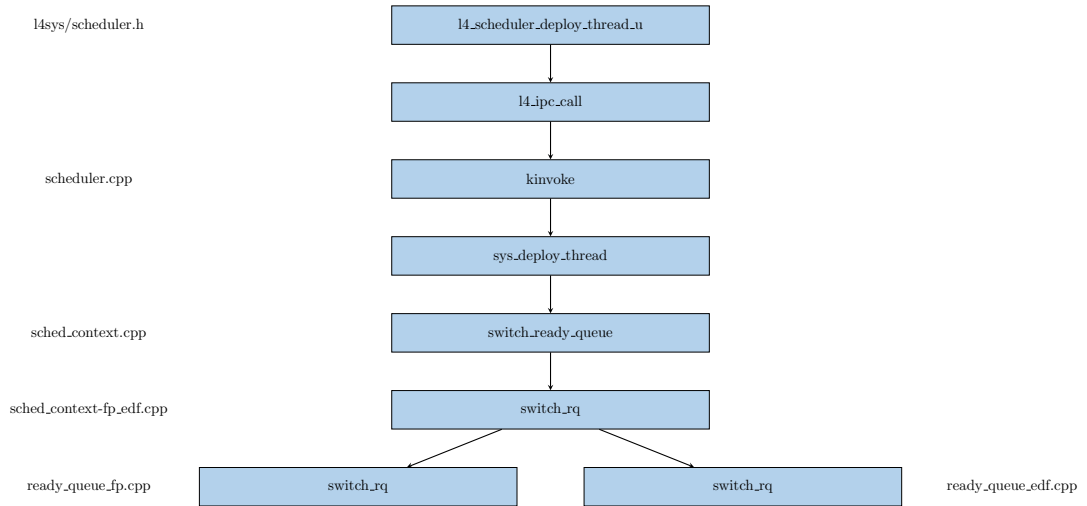


Figure 6.6: Simplified flow of Fiasco.OC and Genode interface for Deploy

system call function (i.e. *l4_scheduler_deploy_thread_u*) are provided respectively (see listing A.11 and A.12). The functionality however is similar to other system call functions in the sense of preparing the relevant messages for the IPC. In contrast to the extension in section 6.2.2, in this case the scheduler object of the kernel (i.e. `L4.PROTO.SCHEDULER`) handles the request.

The scheduler object of Fiasco.OC routes the IPC call to the corresponding handler method *sys_deploy_thread* which was newly created. The process hereby is similar to the presented approach in section 6.2.2. Considering the provided operation code (an overview about supported operation codes can be seen in listing A.13), a corresponding method is called from the *invoke* function of the scheduler object (see listing A.14). In this case, the newly added *sys_deploy_thread* function as shown in listing A.15 is called. This function creates a new ready queue which is based on the actual ready queue from the current scheduling context and the thread which should be deployed. In a later step, the newly created ready queue will be swapped with the current ready queue. To guarantee, that the correct ready queue will be switched, a metric attribute (i.e. priority or deadline) which identifies the corresponding scheduling context will also be assigned into the list. The actual switch of the ready queues is provided by the *switch_ready_queue* method in the scheduling context.

Within the scheduling context, the *switch_ready_queue* function as shown in listing A.16 uses the actual function from the *Ready_queue_base* class. Considering the metric as the second element of the list, the *switch_rq* function (see listing A.17) determines which concrete ready queue, either Fixed Priority or Earliest Deadline First, will be replaced by the new list. As shown in the last given listing A.18, the implementation of *switch_rq* on the example of the *Ready_queue_fp* class reuses the *requeue* function of Fiasco.OC to deploy the new list thread by thread.

Beside the extension for providing a potential deployment functionality from user-space, there are some extensions within the scheduling context made which provide

additional information for later analysis of the system state. In the concrete case the information is about the ready queues themselves. The functions are either capable to get the current state of the ready queue (*get_rqs*) and to get information about potential finished threads within a queue (*get_dead*). The latter is required due to some restrictions caused by the tracing framework of Genode which will be described later.

This section has described the extensions made in the Fiasco.OC kernel, especially the addition of a new scheduling strategy, the extension of timing information per thread, and the creation of a deployment interface. Within the next section the focus lies on the user-space components rather than the kernel.

6.3 Operating System Framework Components

This section will describe the relevant software components which are required for realizing the outlined concept of a flexible task management within the Genode operating system framework (see figure 6.7). First, a network software component which is responsible for the communication with other systems will be described (see section 6.3.1). After this, software components for gathering the relevant system information from the kernel as well as related software components processing these information will be outlined (section 6.3.2). As a central part, the software components which are used to control the system adaptation process will be described in more detail afterwards in section 6.3.3. This section will close with the relevant software components for loading and deploying a set of tasks (see section 6.3.4).

6.3.1 Component for Network Communication

The user-space component for network communication (or simple: network component) is basically realized as a stream socket server (see listing A.19) in combination with a customized protocol. Genode already supports an OSI compliant network stack which is used to implement the server functionality on top. The network component, hereby, fulfills two functions:

1. Input information is processed and (if required), is forwarded to a target component within the system
2. Output information from the system (e.g. monitored data) is sent to other systems

For both cases (i.e. input and output), a customized protocol was developed which is based on so called magic codes (see table 6.2) which are used to remotely control the system. The protocol hereby supports the outlined adaptation workflows as described in section 4.4:

- Adding a new application to the system
- Optimizing a current system state
- Monitoring a current system state

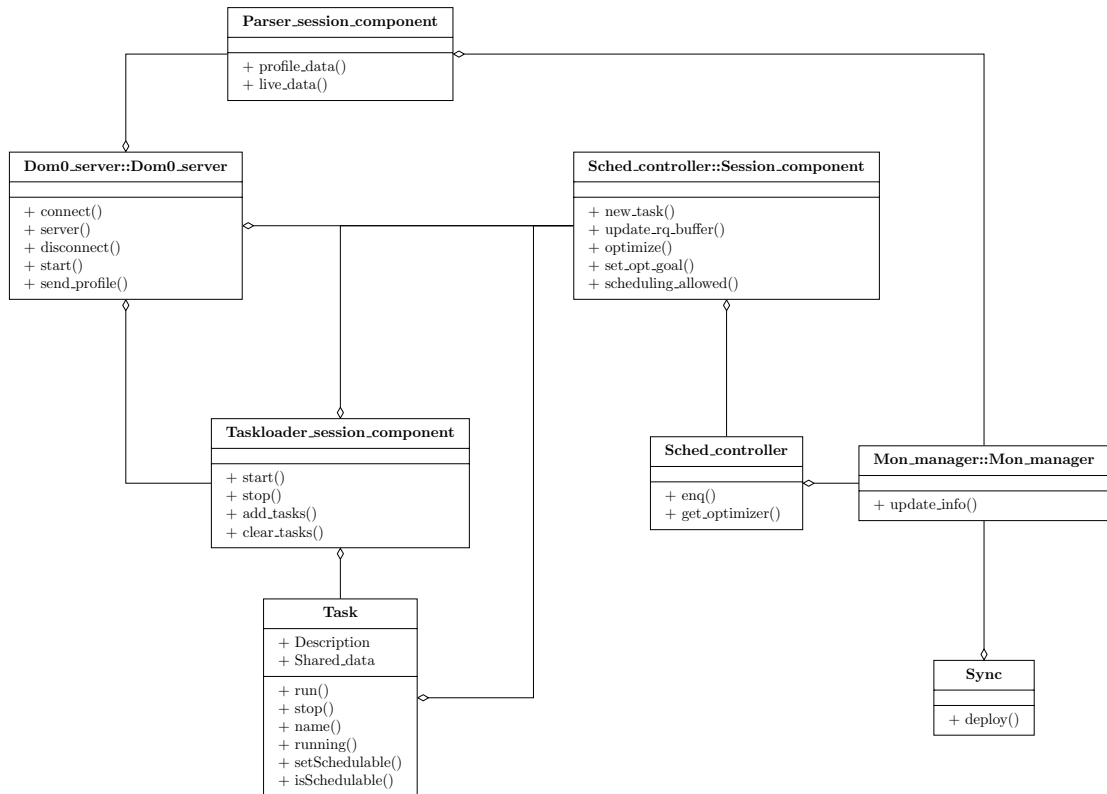


Figure 6.7: Collaboration Diagram of several classes for the flexible task management in Genode

Code	Description
SEND_DESCS	Packet contains task descriptions as XML
CLEAR	Clear and stop all tasks currently managed on the server
SEND_BINARIES	Multiple binaries are to be sent
GO_SEND	Binary received, send next one
START	Start queued tasks
STOP	Stop all tasks
GET_LIVE	Request live info as xml
GET_PROFILE	Request profiling info as xml

Table 6.2: magic codes provided by the network component

The protocol, hereby, poses a simple key value pair mapping where a magic code is assigned to a distinct method which handles the request accordingly. Some magic codes however require the sending or receiving of additional information. For this purpose, a data file is used which contains the required information. In case of optimizing the current system state, the optimization goal is transferred within a data file rather than a magic code. This keeps the foot print of the control protocol small. By monitoring a current system state, monitored data are encoded in a data file which can be sent back to the client. For adding a new application to the system, however, a further file needs to be sent to the system - a binary file which represents the compiled application. In this case a data file is used to provide additional (i.e. meta) information about the application like input parameter or timing attributes.

The central *serve* method of the network component as shown in listing A.20 demonstrates the relevant structure for consuming the earlier described magic codes. Each case within the main server-loop represents one distinct action (e.g. *SEND_DESCS*) which is handled in this distinct branch. For example, task descriptions and task binaries are provided to the taskloader component (see below) which is responsible for the loading of the tasks.

After describing the component for network communication and their basic workings, the next section describes the relevant user-space components to get distinct information from the system.

6.3.2 Components for Information Gathering

This section will provide an extension of Genode's tracing framework to enable the collection of timing related attributes from the underlying kernel. This section therefore corresponds to section 6.2.2, with the difference that it describes the user-space view of the collection process (i.e. stops at the *Platform_thread* class). Most of the presented software components are required for the collection of information. Considering the monitoring of a current system state, the collected information need to be prepared for the transmission via the network component. This section therefore closes with details about a related serialization process of the collected information.

object is designed as general as possible whereas for the context of this thesis the majority of stored elements within the *Monitoring_object* represent task related information (e.g. timing attributes).

Each user-space component (e.g. controller) which requires distinct information about the system, requests the service of the *Mon_manager* class which in turn provides this service via its *update_info* function (see listing A.22). First and foremost, each component which needs system information provides the *Mon_manager* with a dedicated dataspace (i.e. shared memory) out of its own memory quota (due to the capability regulations). This shared memory is attached to the monitoring memory via memory mapping. After that, the *Mon_manager* establishes a connection to the tracing service. For storing the collected information about user-space components, a memory buffer with tracing subjects will be reserved. Each user-space component consists of several threads which are sharing a common session name (i.e. name of the user-space component). Each tracing subject corresponds to one single thread. Basically, *update_info* iterates over all tracing subjects and fills the corresponding *Monitoring_object* of each tracing subject respectively. Based on the example of getting a thread's execution time, the common process for getting system information will be outlined.

By establishing a connection to the trace service, an object of the *Trace::Session_component* class will be instantiated. This class provides a set of functions which can be used to get the distinct information about the cpu, the memory or the scheduler. To get the actual information, *Trace::Session_component* requests the *Trace::Subject_registry* which holds all tracing subjects of the system. Basically, the registry provides a search of the subject in question and calls the corresponding method of the tracing subject. For example, a call of the *cpu_info* method results in the *info_cpu* method (see listing A.23) provided by a *Trace::Subject*.

The main purpose of the *Trace::Subject* class is to group the incoherent information (i.e. scattered anywhere in the system) into a uniform representation. Each instance of the *Trace::Subject*, therefore, consists of tracing sources (i.e. where the information comes from) and the corresponding groups of information (e.g. *Trace::CPU_info*). In the case of getting the execution time, the *info_cpu* method gets the information by calling *info* method of a related source. Further information (e.g. affinity) is gathered and stored in the same way. As a final step, all collected information about the cpu are grouped in a *Trace::CPU_info* class (see listing A.24).

Investigating the *info* function within the *Trace::Source* class, the listing A.25 shows the basic process about getting the relevant information. As expected, a task's execution time belongs to a kind of dynamic information which is changing over time (i.e. in contrast to static information which are fixed). The internal structures *Info*, *Dynamic_Info*, and *Static_Info* are used to store the several types of information (e.g. labels, priorities, time related parameters). As a result the function returns an *info* object with the accumulated information of dynamic and static data elements.

In the case of getting *execution_time*, the corresponding implementation of *dynamic_info* can be found inside the *Cpu_thread_component* which is responsible for the management of threads within Genode. As shown in listing A.26, the *dynamic_info* function gets its information by calling several methods within the *Platform_thread* object. In case

of getting the execution time, The *Platform_thread* provides a corresponding function with the same name. The concrete method behind the *execution_time* function of the *Platform_thread* is the formerly described *l4_thread_stats_time* kernel function.

Besides the execution time there is other information which is collected during the system execution. These information corresponds to the elements found in the monitoring object which was explained in the beginning. Where it can be seen that not only timing relevant information is collected, but also information for identifying the tasks and their memory usage. All these information together is collected from several places over the system, partly from the kernel and from the user-space. But all information has in common that they are represented in an internal system dependent way and are, so far, not ready to be transferred to other systems.

The *Parser_session_component* is also an user-space component which transforms the internal information data to a serialized object for transferring. Basically, this component uses the already implement functionality of Genode's XML facility. As it can be seen in listing A.27 the parser component prepares a dataspace (i.e. shared memory) where the monitor component can store its information inside. For each traced object, a corresponding XML node is created within the former created XML tree. All information which is provided by the monitoring component is ncapsulated in corresponding XML attributes. This happens for all subjects which should be traced. After that, the create XML structure can be transferred to other systems via the usage of the network component.

In conclusion, this section has provided an overview about the basic monitoring infrastructure for getting distinct information about user-space software components. In a next step, this information can be used to realize a flexible management of such software components. This management approach is based on a dynamic adaptation process (i.e. adding) of running software components. An unconditional adaptation however is not applicable. So, the central user-space component which is used to control the adaptation process of the system will be presented within the next section.

6.3.3 Component for Controlling the System

The adaptation possibilities of the flexible task management need to be controlled and, if required, to be constrained. Both aspects are covered by the user-space *Sched_controller* component. This component is responsible for the required checks and provides its services to other components like the *Taskloader_session_component*. This section describes the implementation in more detail, where an overview about the involved classes can be seen in the collaboration diagram depicted in figure 6.9. Beginning with a basic operation of adding a new task to the system during run-time, subsequent steps and flows will be described including the acceptance test and optimization mechanism.

Basically, adding a new task (i.e. software component) to the system during run-time is provided by the *Taskloader_session_component*. The relevant *add_tasks* function however utilize the *new_task* function of the *Sched_controller::Session_component* which tests if the new task can be seamlessly integrated (i.e. without side-effects) into the existent system. In conclusion, adding a task by the taskloader should be verified by

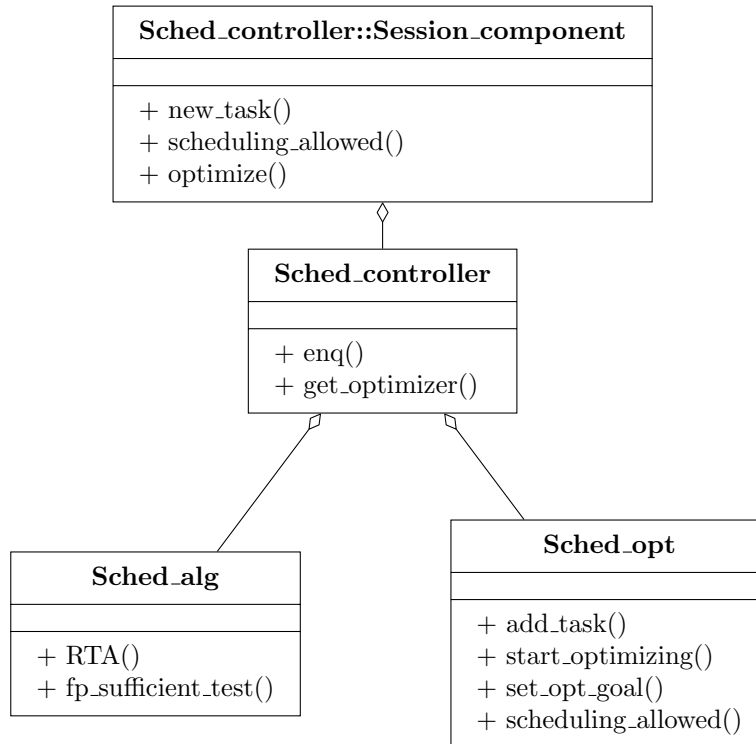


Figure 6.9: Collaboration diagram for Sched_controller

the controller if this task is schedulable within the system or not. Along this analysis process, several tests will be done according to the criticality of a task.

The former function call of *new_task* will trigger the *enq* method within the *Sched_controller* (see listing A.28). Basically, there are two different approaches implemented to test a task. In the first case, an acceptance test is executed to check the schedulability of the new task before its actual execution (relevant tests are in the *Sched_alg* class). The acceptance test is a combination of response time analysis and sufficient test as described in the concept. On the other hand, the task is accepted anyway and an optimization routine modifies the task during its actual execution (implemented in *Sched_opt*). Of course, the optimization process can also be used as an acceptance test (before execution) however this not done in this thesis. By design, the decision if an acceptance test or an optimization should be done for the task depends on its criticality level. Within the context of this thesis, the criticality level is assigned to a corresponding scheduling strategy either fixed priority (i.e. high criticality) or earliest deadline first (i.e. low criticality). The categorisation of a task in high critical and low critical however solely takes place in user-space (i.e. for Fiasco.OCs scheduler treats threads without criticality).

Acceptance Test

The corresponding algorithms for testing if a new task can be accepted by the system are implemented within the *Sched_alg* class. An important aspect for the usage of these functions is the fact that the tasks have to be sorted according to their priorities within the ready queue. Another prerequisite for the algorithms is that the current task set is schedulable without the new task (i.e. initial system is schedulable). Furthermore, each task for itself is schedulable. The implementation details of the algorithms are very exhaustive therefore a description in pseudo code was chosen to explain the basic procedure.

The RTA algorithm is an iterative approach to calculate the response time of each task within a task-set. The calculation of the response time continues until the response time doesn't increase anymore. The overall task-set is only schedulable if for all of its tasks the condition $R_i \leq D_i$ is given. Choosing an appropriate initial value (i.e. in this case worst case execution time) for the response time reduces the number of required iterations (i.e. tests) for one task. Due to the fact that in the given preemptive system a new task only influences tasks with the same or lower priority, the set of to be tested tasks can be further decreased.

The response time analysis (RTA) as shown in algorithm A.1 covers the several cases in which each task and ready queue can be. The actual calculation of the response time is done in the *cmp_response_time* function (see algorithm A.2) according to the equation 6.1.

$$R_i^{n+1} = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j \quad (6.1)$$

There are the following cases which need to be checked by the RTA algorithm. First, if the task-set is empty the task is accepted without any further tests (i.e. under the given condition that each task alone is schedulable). Second, if the priority of the new task is lower than all other tasks within the task-set, the response time for the new task itself need to be calculated. In all other cases, the start point for the response time calculation has to be found where the priority of the new task is used to find the first task (within the task set) with equal or lower priority. For all tasks after this starting point, the response time has to be recalculated. If all tasks have passed the test, the new task can be added to the system. The RTA algorithm is an exact but expensive test with an exponential run-time behavior. In the context of this thesis therefore a sufficient but more efficient test is also provided.

For the realization of a sufficient test (i.e. doesn't find all schedulable solutions) for fixed-priority task-sets, the schedulability test of Bini [19] was chosen (see algorithm A.3). Again, for each task within task-set the condition $R_i \leq D_i$ need to be given to pass the test. The calculation of the response time follows according to equation 6.2.

$$\forall_i R_i^{ub} = \frac{C_i + \sum_{j=1}^{i-1} C_j (1 - U_j)}{1 - \sum_{j=1}^{i-1} U_j} \leq D_i \quad (6.2)$$

The run-time behavior of the implemented algorithm depends only on the number of tasks within a task-set (i.e. $O(n)$). This is achieved by accumulating the sums within the fraction from task to task. Due to this approach the algorithm needs to iterate over the task-set for the required checks only once. This adds a new task similar to finding an adequate position within the task-set (cp. RTA starting point). “The new task has to be proven before the upper bound for the response time of the first task with a lower priority is calculated. If the upper bound is lower or equal than the deadline for every task, the task-set is schedulable.” [53]

Optimizing

The optimizer algorithm is provided by the *Sched_opt* class. The *add_task* function is responsible for adding a task to the set of tasks within the internal task set used by the controller for a later optimization. It is mandatory, that each task which is subject of a potential optimization is added via this method. The entry point for the central optimizing routine is *start_optimizing*. This function is called during the actual execution of a task.

The implementation of the *start_optimizing* function consists of several hundreds lines of code due to the fact that the overall optimization process covers several use-cases. For a better overview, figure 6.10 provides a flow graph diagram for the optimization process as a whole.

As the first step, the optimizer queries the monitoring component via *update_info* for any system changes (i.e. task changes). The tasks and threads are separated from each other and there exists no direct information about which thread belongs to which task. A separate matching therefore is required which assigns the threads to their tasks by using a label as identifier. As a result, there exists a list of tasks and their threads which are interesting for a potential optimization. This step is commonly known as data query in the optimization process, where the analysis of the queried tasks will be discussed in the following.

The analysis process of the optimizing function within the controller component is departed in several steps (see figure 6.10). These steps are used to further investigate the former queried tasks to adapt in accordance to their states the optimization process. In principal, it can be differed between a successful execution and a non-successful execution of a task with its corresponding threads. However, both branches are based on some common used functions which will be shortly presented in the following list:

- **Modification of value** The optimization value is a central attribute which is considered by the optimization. Each optimization step requires a corresponding modification (i.e. increase, decrease).
- **Update functions** During the optimization process, used structures and variables need to be updated. The attributes in this case are utilization, permissions and tasks.

For both cases (i.e. successful and non-successful), the presented functions are used in different combinations. Starting with a successful executed task (i.e. regularly terminated), its value will be decreased which allows other tasks in a next execution cycle to be executed. Accordingly the utilization, the dispatched flag and the scheduling permission will be updated. All in all, the task is set to be successfully executed. The more complex part (i.e. in the sense of checking) is if a task (or its jobs) were not successfully executed. When this happens it need to be clarified why not. In this case there are basically three reasons which are checked, namely the task was not allowed to be scheduled, a task missed its deadline or a task was killed. Where in the last both cases (e.g. deadline missed or killed) another request to the monitoring component needs to be made. In this case however not the actual task list will be requested, rather the list with terminated tasks (i.e. RiP list).

This section has provided details about the general controlling process via the *Sched_controller* component. Two different testing approaches has been provided which allow the test before or during a task execution. In a last step, the next section closes the gap and provides the missing parts for loading and deploying of tasks.

6.3.4 Components for Loading and Deploying of Tasks

In case of adding a new tasks or optimizing a current system state, tasks need to be integrated or removed from the system. These changes are mandatory in user-space as well as in kernel-space. For the former, a component (i.e. task loader) is responsible for the loading of arriving tasks. The system state needs to be consistent between user-space and kernel-space. Each change within the task set however makes a synchronization between the user-space and kernel-space mandatory. This synchronization is done by another component with the corresponding name. The following describes both components, especially their implementation in more detail beginning with the taskloader.

The taskloader is responsible for the following aspects:

- manages tasks (create, delete)
- parses arriving task descriptions
- stores task binaries in its own dataspace

An important detail about the taskloader stems from the strong parent child concept introduced by Genode. Every task which will be added during run-time needs to be a child of taskloader (as parent). As mentioned earlier, only a parent has the full capabilities to manage its child accordingly. Thus, the taskloader is allowed to create or delete its children (application tasks). A similar component which allows the creation of tasks within in Genode is the init process. The taskloader memes a second level init process in this way, because there are no other possibilities (except through huge modifications of init) to control the children (i.e. deployed tasks) in the intended way.

Adding new tasks to the system follows a simple routine in which the corresponding task description will be parsed and relevant information will be extracted (see list-

6.3 Operating System Framework Components

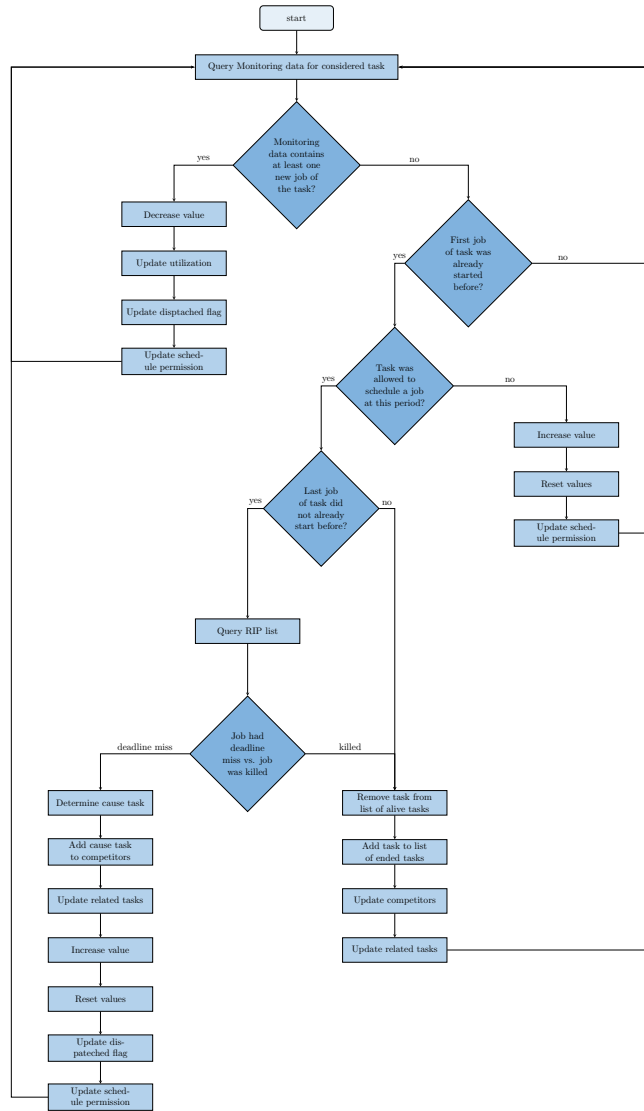


Figure 6.10: Flow Chart for Optimization Procedure according to [112]

ing A.29). For each task within the task description, different tests will be performed to check if the new task is schedulable according to the chosen strategy.

If the controller component decides that the resulting task set is schedulable for the current system state, the new task set (which is only existent in user-space) needs to be deployed within the kernel. This deployment is done by the synchronization component which allows a fine grained determination when the right point in time for a synchronization will be. The synchronization component therefore provides a *deploy* functionality which maps the *deploy_queue* function of a platform thread in its simplest form. In this case the task set will be instantly deployed within the kernel without further checking of possible hazardous results. Within the platform thread the former

explained `l4_scheduler_deploy_thread` method is responsible for the actual exchange of the queues.

With reaching the end of this section, all relevant user-space software components which are used to realize a flexible task management have been described. The next section will cover possible limitations of the current implementation.

6.4 Limitations of the current Implementation

This section will describe some implementation constraints of the so far realized user-space components and kernel objects. All constraints are grouped around a central problem, namely the representation of a system state (i.e. set of ready queues) in user-space via a dedicated buffer structure. This problem concerns all of the following actions:

- getting system information
- adding a new task to the system

Beginning with the limitation during the process of getting system information, user-space components as well as kernel objects will be addressed. It follows the limitations during the addition of a new task to the system.

6.4.1 Getting System Information

During the process of getting system information, several limitations can be identified. It starts with a central problem in the notification mechanism about system changes. This affects consecutive monitoring and analysis processes. Furthermore, there are some limitations from the used tracing framework which will be described. Finally, the cause and the consequences of identity translation (e.g. matching) between user-space components and kernel objects will be outlined.

Beginning with the exchange of information between kernel and user-space, some limitations need to be remarked. Due to its simple structure, an `utcb` is merely an one dimensional array, all information needs to be simple (i.e. no complex data types are allowed). In best case the information is a scalar unit, like in this case a timing attribute. Another limitation is the high level of abstraction from the kernel upwards to the user-space. So, many levels need to be intersected to transport the given information out of the kernel into the user-space. This could be a potential problem if the reaction of a system will be strongly influenced by a great number of information exchanges (or exchanges with few but huge information sets). Furthermore, the internal routing through a single `ipc_call` and a required resolving considering the given capability can slow down the overall performance of the system.

A central problem within the so far described implementation covers the tracking of current system changes. There is no notify mechanism similar to an interrupt service routine concept which can be used to call distinct functions (e.g. optimizing) at certain events (e.g. task exist). If a component needs to get continuous information about the

6.4 Limitations of the current Implementation

system state it needs to call the *update.info* in a busy waiting-like fashion. A possible out-of-sync between the current system state and the representation in user-space however is possible. This problem influences the overall information gathering process for the internal components as well as the external systems (e.g. for a later analysis). As an example, the networking component undergoes several changes to deliver an adequate system state back to another system (i.e. work station). The first version to transfer information about system changes between systems was an transaction-based process where the system in the active role starts the transfer and commits the transaction by requesting an update (i.e. *profile_data*) at the end. The problem with this approach is that during start and commit, monitoring data vanishes due to the fact that only the recent timing information (i.e. dynamic task information) can be traced (i.e. storage constraints on an embedded system). Indeed events (e.g. task finishes) can get lost because the point in time where the information is updated doesn't correlate with the point in time when an event arises. Improving this situation, the new implementation was time-based rather than event-based which means that the system was able to transfer its system-state at any point in time (i.e. *live_data*). The system which requests the actual system state however needs to poll the information in a short period of time. This improves the situation considering the consistent information gathering (i.e. more information about corresponding events). As a result, more events can be traced dependent of the chosen time interval. But as a restriction the interval couldn't be set to a number where all system changes could be traced. So, the chance to miss certain events was reduced but not eliminated. The last implementation improvement taken so far, was to transform the system in question to an active sender. This is realized via a push service, where the system itself delivers its state pro-actively to other systems. As a side-effect of this solution, each notification about a system change needs to be routed through the network component.

Beside the missing notification support there is another limitation rooted in the used tracing framework. In the case a task will be killed via a remote kill command, the task will be removed from the tracing framework. This causes that the monitoring component to be unable to trace this task anymore (i.e. get information). Considering the information about the finishing time of a task, this is not optimal. It needs therefore to be a solution where the information about killed tasks is still accessible via the monitoring component, even the task itself is not presented anymore within the tracing framework. For the solution, a separate list (*Rest in Peace - RiP*) was introduced. With the assist of this list a task gets recorded shortly before it is killed. The last existent information (i.e. before killing) remains as a snapshot within this list and can be requested as usual.

For the last limitation concerning the gathering of information, an understanding of how Genode and Fiasco.OC handles their scheduling priority values is required. "The scheduling in Genode is for the most part handled by the chosen kernel. Genode merely abstracts the common scheduler parameters, namely priority and CPU core affinity, and makes these values available to the framework. However, how these parameters affect performance greatly depends on the chosen kernel." [64] As it can be seen in figure 6.11, Genode maps its available priorities to a subset of Fiasco.OC's available priorities. A matching which translates Genode priorities to Fiasco.OC priorities is therefore required.

Genode		High			Low			
Available Priorities	0	1	...	32767	32768	...	65534	65535
num_levels = 2	0			32768				
Fiasco.OC								
Available Priorities	127	126	...	64	63	...	1	0
Effective Priorities	64			0				

Figure 6.11: Genode and Fiasco.OC priority values [64]

The loose coupling of user-space and kernel-space also enforces the demand of a matching mechanism. This matching mechanism is not a single function within the system rather than distributed over several places within the system (i.e. whenever a analysis of a task set is required). The *Monitoring_object* is the central structure within this matching process. However, the fact that Genode and Fiasco.OC choose their identification number of a task/thread independently and randomized leads to a problem. This means there exists no direct translation between Genode’s identification numbers and Fiasco.OC’s identification numbers. The translation is done via the help of an unique label name which is assigned to each task. Each *Monitoring_object* therefore implements a label, *genode_id* and *foc_id*. Without a doubt, a matching process introduces a certain overhead within the system.

After outlining the limitations during the information gathering process, the next section covers the limitations arising if a new task will be added to the system.

6.4.2 Adding New Tasks to the System

The flexibility by adding a new task to the system during run-time is constrained by some limitations which will be covered within this section. There are certain prerequisites which are required to successfully deploy a new (external) task within the system. On the other side, the capability mechanism constraints a straightforward implementation of the synchronization component.

Before it is possible to add a new task to the system, some prerequisites need to be fulfilled. This starts with the selection of the combined scheduling context (i.e. fp/edf) during compile time. This configuration is fixed as long as the system is active and is valid for all cores. The active schedule per core however can be set within Genode’s configuration file (i.e. run files) at run-time. The affinity, priority and deadline attributes of each task are encoded in the task description which is send to the system during run-time. A check if new tasks corresponds to one of both scheduling strategies is done in user-space. One of the biggest drawbacks of the chosen combined scheduling approach is its worse scalability. For two strategies, the management overhead is acceptable but increases with every additional scheduling policy. Foremost the management of threads (i.e. enqueue) gets more complex which can result in a potential performance degradation.

The synchronization component is meant to control the point in time when the modified task-set gets deployed into the kernel. Required capabilities to allow the synchronizer to work in such a way, however, are missing. As an intermediate solution, tasks will be

deployed via the taskloader (i.e. 2nd level init) which holds the distinct capabilities. So the control of the kernel scheduler from the user-space doesn't work as expected. Therefore the only task sets which are allowed to be executed by the scheduler are already deployed ones. For example there exists no control mechanism within the user-space to force the instant execution of a task independently from the actual system state. In the current implementation, the user-space components prepare the new task set and hands over the control to the kernel scheduler which executes the task set as a whole. Enabling the synchronize component to deploy (and control) the task set from user-space enforces the following steps.

1. Getting new "deploy" capabilities from the kernel
2. Modifying taskloader, controller and synchronizer component to get an adequate deploy cycle
3. Enhancing the synchronizer component to realize the conceptual ideas (e.g. deploy thread in certain point in time)

Moreover, the current deployment function only works with a fixed priority preemptive scheduling strategy.

Conclusion

This chapter has provided the extension of a micro-kernel base operating system. It therefore has described which functionality is already provided by Fiasco.OC and Genode. The main part of this chapter has covered the required kernel-space and user-space modifications for realizing a flexible task management. The chapter has closed with certain limitations of the current implementation. The next chapter will present the evaluation results of the developed system.

7 Evaluation of the Flexible Task Management

This chapter will present different evaluation results which are sampled through a number of testing scenarios. Each scenario is used to proof distinct operational conditions of the flexible task management. The first testing scenario will evaluate (section 7.1) a possible usage of the operating system within an automotive context. The second scenario (section 7.2) is used to analyze the consolidated architecture as well as the realized operating system. In this chapter considered testing aspects correspond with the outlined contribution item number five in chapter 1.

7.1 Scenario I: Reliable Autonomous Driving under Adaptation

This section will demonstrate the usage of the developed approach in the context of an automotive system. The overall test setup as well as the autonomous driving scenario are described in section B.2 in the Appendix B. The concrete contribution which will be covered in this section is a case study of an live update/installation within an automotive driving scenario (hybrid simulator test bed) during operation (cp. contribution 5e) .

The chosen case studies therefore simulate both an installation and an update process of an application software component during an autonomous driving maneuver. In both cases, an adaptation of the current system during operation will be performed. In section 7.1.1, the initial installation process of a new software component will be described. After that, section 7.1.2 will detail the update process of an application component.

7.1.1 Installing an Application Component During Operation

The installation process which will be outlined in this section deploys a application component to the target system. As an example application, the network communication component, labeled as *mbl_adapt*, of the motor control subsystem was selected. This component is primarily used to receive incoming motor control commands over the network. With the help of another software component (i.e. *mbl_client*), the incoming network commands are mapped to concrete function calls which are used to control the engine. If the *mbl_adapt* application is missing from the system, the communication is interrupted and the engine doesn't get any movement commands. The idea to proof a successful installation is to observe that the engine gets from an initial state (i.e. engine stands still) to an active state (i.e. engine starts).

The principal setup can be seen in figure 7.1. There are several inter-connected control units (i.e. Pandaboards and Raspberry Pi) re-creating a vehicle electrical system. Each

7 Evaluation of the Flexible Task Management

control unit is equipped with the developed operating system. The Pandaboard-based control units represent the “logical” devices which are comparable to domain controllers and consist of a dual-core system (In the following labeled as *Core0* and *Core1*). The network communication as well as certain management tasks will be executed on these boards. In contrast, control units which are represented by the Raspberry Pi single-board computer are responsible for the direct control of the connected hardware (i.e. sensors and actuators). These boards are also able to communicate via a network interface but provide the concrete “control” interface of the connected hardware.

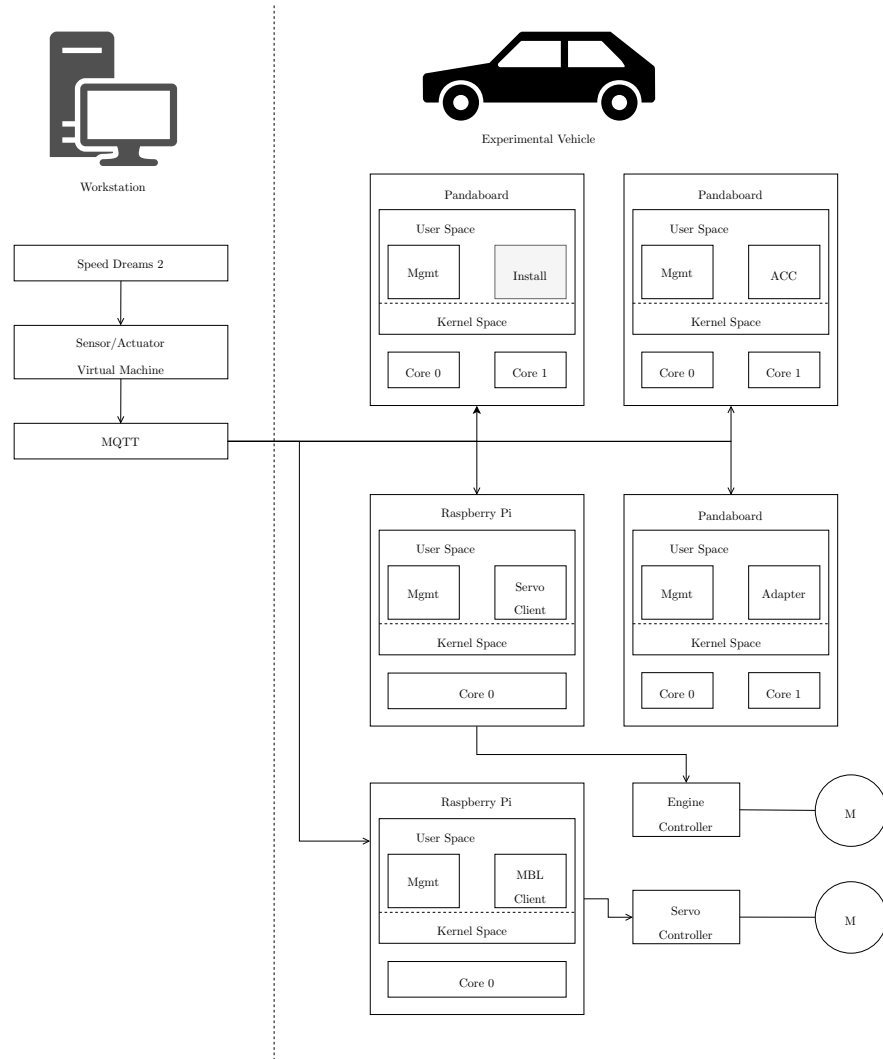


Figure 7.1: Installation Case Setup Car and Workstation

The control unit with a gray box, labeled with “Install”, marks the target unit on which the *mbl_adapt* application would be installed. This, for the installation scenario, prepared control unit executes the developed operating system with all flexible task

7.1 Scenario I: Reliable Autonomous Driving under Adaptation

management components on *Core0*. The to be installed component will be deployed on *Core1* of the board which doesn't execute any other software components. Both cores of the control unit are managed by fixed priority preemptive scheduling strategies. An acceptance test is therefore automatically chosen by the *sched_controller* component. The installation scenario follows an automated testing routine which consists of a script (see listing B.4) and a corresponding task description (see listing B.5).

The following installation sequence is encoded in the log files B.6, B.7, B.8, and B.9 provided in Appendix B.

Time (mm:ss)	Component	Description
00:20	<i>mbl_client</i>	connected and ready for receiving commands
00:28	dom0-HW	connected and waiting on receiving task description and task binary of <i>mbl_adapt</i> application
03:31	dom0-HW	receives task description and task binary of <i>mbl_adapt</i> application
03:32	taskloader	starts task
03:37	<i>mbl_adapt</i>	connected and ready for receiving commands
03:39	<i>mbl_client</i>	receiving motor control commands

The successful installation of the required component can be seen by the starting movement of the engine. Figure 7.2 shows the result before and after the installation. Between the standing engine and the rotating engine there are several minutes elapsed.

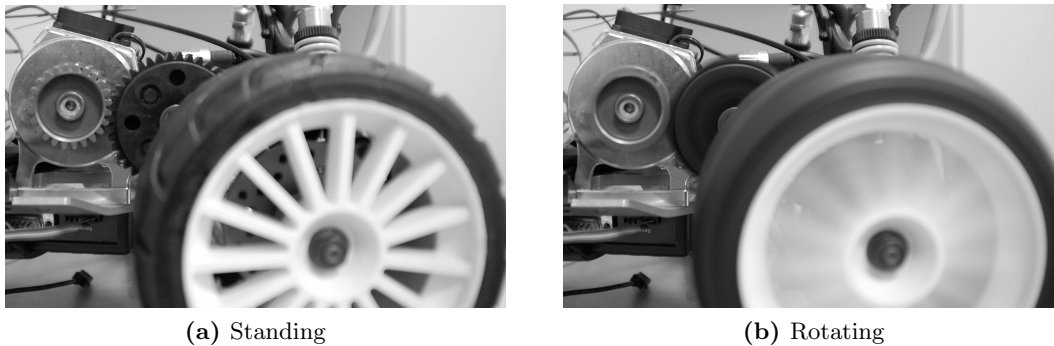


Figure 7.2: States of Motor Movement

This section has provided an installation process of a missing application component on one of the control units within the vehicle board network. The successful installation shows the principal feasibility of a flexible task management within the context of an automotive system. All processes are executed during the vehicle's operation at run-time. As an extension of this test scenario, the next section provides an update scenario where an existing software component on one core gets updated.

7.1.2 Updating an Application Component During Operation

This section will describe the other use case within the autonomous driving scenario, namely an update of an existing application component. For this use case an application which calculates the fractorial portion of the number pi will be used. Consecutive updates of this component are used to increase the number of to be calculated decimal places. This use case principally demonstrates the application of an update within an automotive system during operation.

The setup for this scenario can be seen in figure 7.3. In contrast to the installation case, the target control unit is hereby executed in a virtual environment due the possibility of simulating four cores (i.e. the hardware control units exclusively provide two cores).

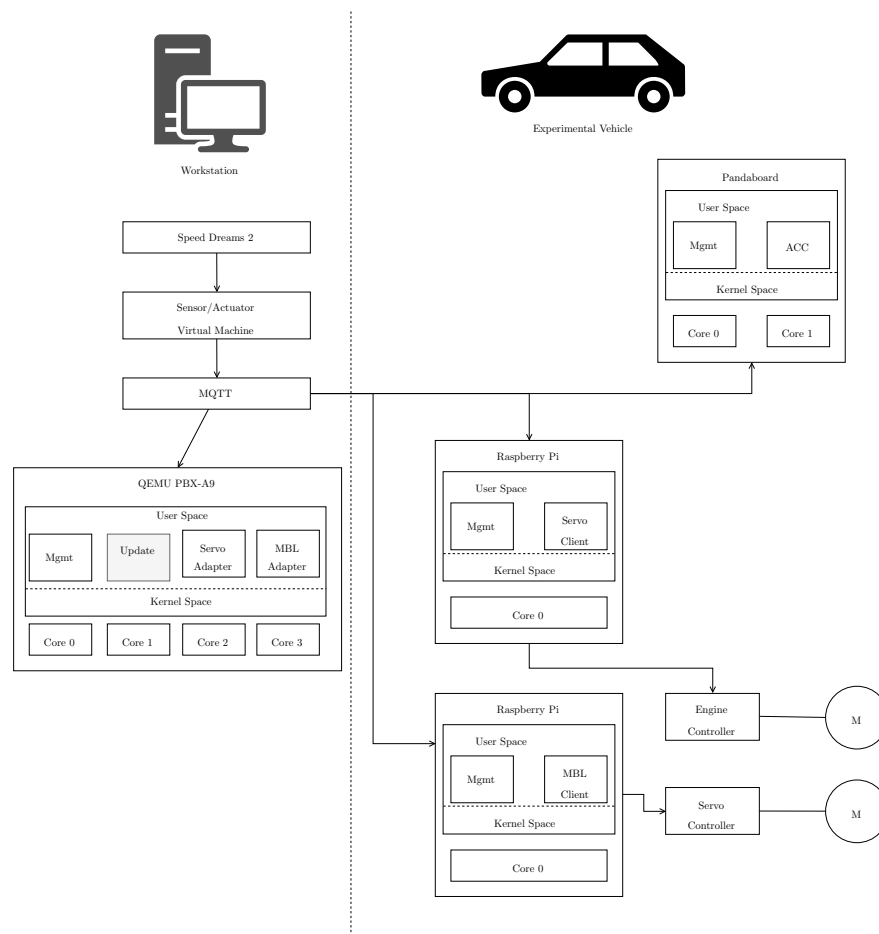


Figure 7.3: Update Case Setup Car and Workstation

Within the virtual environment, a quad-core system is simulated which executes the developed operating system with the following configuration:

7.1 Scenario I: Reliable Autonomous Driving under Adaptation

Core	Description
<i>Core0</i>	Executes the operating system with management components
<i>Core1</i>	Executes the application component which is subject to an update
<i>Core2</i>	Executes the network component for controlling the engine
<i>Core3</i>	Executes the network component for controlling the steering and braking

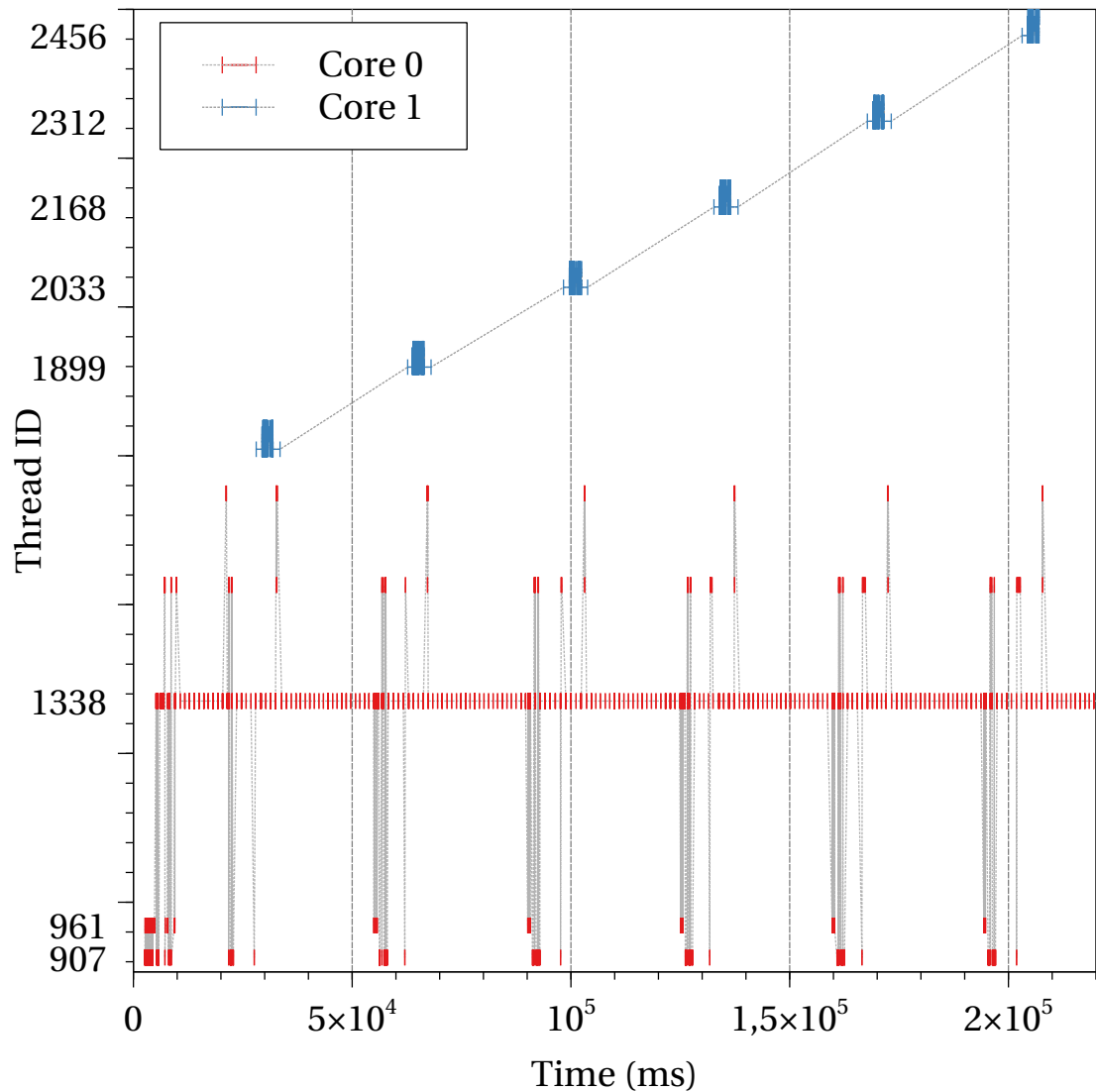
All cores are managed by fixed priority preemptive scheduling strategies. The update process is controlled by the workstation via a testing script (see listing B.10) and a corresponding task description (see listing B.11). Each update is done with a distinct application version (i.e. varying input parameter). For example, the initial version of the application calculates the number of Pi according to a given numeric argument (e.g. 2 decimal places). An update of this application is considered as changing this numeric argument (e.g. increase it by one). Each application is allowed to finish its calculation (i.e. no kill operation).

The resulting sequences on *Core0* (colored in red) and *Core1* (colored in blue) are depicted in figure 7.4. This figure shows the following tasks:

FOC ID	Component
907	Taskloader
961	Parser
1338	Monitor
1533	Schedule Controller (Controller)
1687	Dom0-HW (Network Communication)
>1761	Pi Application

An update is hereby periodically processed for the purpose of testing. As it can be seen there is an interplay of relevant management components on *Core0*. The monitor component itself periodically observes the system and serves as information provider for the controller and the taskloader component. In general, deploying a task (i.e. updating) can be seen as a pattern within the figure: Dom0-HW communicates with the taskloader and the taskloader itself communicates with the controller. This represents exactly the wanted system behavior. At $2.9e4ms$ the taskloader finally deploys the task into the system and after $3.3e4ms$ the dom0-HW registers the exit state of the deployed application. At $5.1e4ms$ the parser gets its information from the monitor component and prepares this information for transferring. At the point $5.11e4ms$, the taskloader again deploys an application (i.e. updated version). Because the testing script triggers a periodic update script, there is also the same system behavior identifiable over time.

This subsection has provided an update scenario, where an application component is updated during run-time. Both scenarios which were covered in this section has given an overview about the overall install and update capabilities of the developed system. The next section therefore will outline a more in detail analysis about certain single steps which are parts of the overall process.

Figure 7.4: Update of component on *Core1* over time

7.2 Scenario II: Testing System Properties using Artificially Generated Task Sets

This section will present the results for testing the operating system in an software-in-the-loop like setting. The operating system, thus, is executed on a distinct control unit which is connected to a workstation. Within this setup, artificially generated task sets will be used to test the distinct system properties. First, the separation between software components on a multi-core platform (cp. contribution 5a) will be investigated in section 7.2.1. This is followed by of the co-existent scheduling approach in combination with an optimization process for a robust execution (see section 7.2.2) that

demonstrate the feasibility for contribution 5b and 5c. This section will be closed with first timing measurements (cp. contribution 5d) about the distinct adaptation phases (see section 7.2.3). The overall test setup for this scenario is outlined in Appendix B.

7.2.1 Support for the Separation of Software Components

One of the principal aims by using a multi-core based hardware platform in the context of this thesis is to provide a certain separation of the adaptation control components (i.e. management) from the application components (i.e. actual execution). A central aspect which need to be investigated is therefore the interplay between the software components across cores boundaries. The following tests are provided on a dual-core system where the first core is labeled as *Core0* and the second core as *Core1*.

Of course, separating different user-space components heavily depends on the resource management capabilities which are provided by Fiasco.OC and Genode. The following tests therefore focus on the interplay of user-space components by using services which are directly provided by Genode (i.e. flexible task management components will be excluded). The first test utilizes a periodically waiting task, labeled as *idle*, executed on *Core1* which uses a timing service, labeled as *timeout-scheduler*, on *Core0*. The plot 7.5 depicts the interplay between both cores. The bars colored in blue represent the timing behavior of *Core1*, whereas bars in red are indicating the behavior of *Core0*. The numbers which are labeled along the y-axis represent the process ids executed on each core:

- entry-point: 263, 220, 13
- signal: 274, 231
- idle: 198, 191
- timeout-scheduler: 242
- core and system: 7, 9

The green boxes mark certain points where an interplay between the components takes place. At first *idle*'s main thread (191), entry-point thread (263) and signal thread (274) are created on *Core0* and will be moved to *Core1*. This process is marked via the two green boxes on the left side. If *idle* starts its execution, it continuously triggers a signal (274) to wait for a certain amount of time. This signal is received by the entry-point (220) of the *timeout-scheduler* component. This component starts its execution (242) and exclusively utilizes the core for an amount of time which equals the waiting time of *idle*. This circumstance is marked with the third green box. Due to the fact that *idle* is waiting periodically, the remaining plot depicts exactly this repeating process. As a first insight, there is a remarkable interplay between both cores. In a worst case scenario, this interplay can lead to a over-utilization of *Core0*. As a next step, the condition under which an interplay can be reduced needs to be investigated.

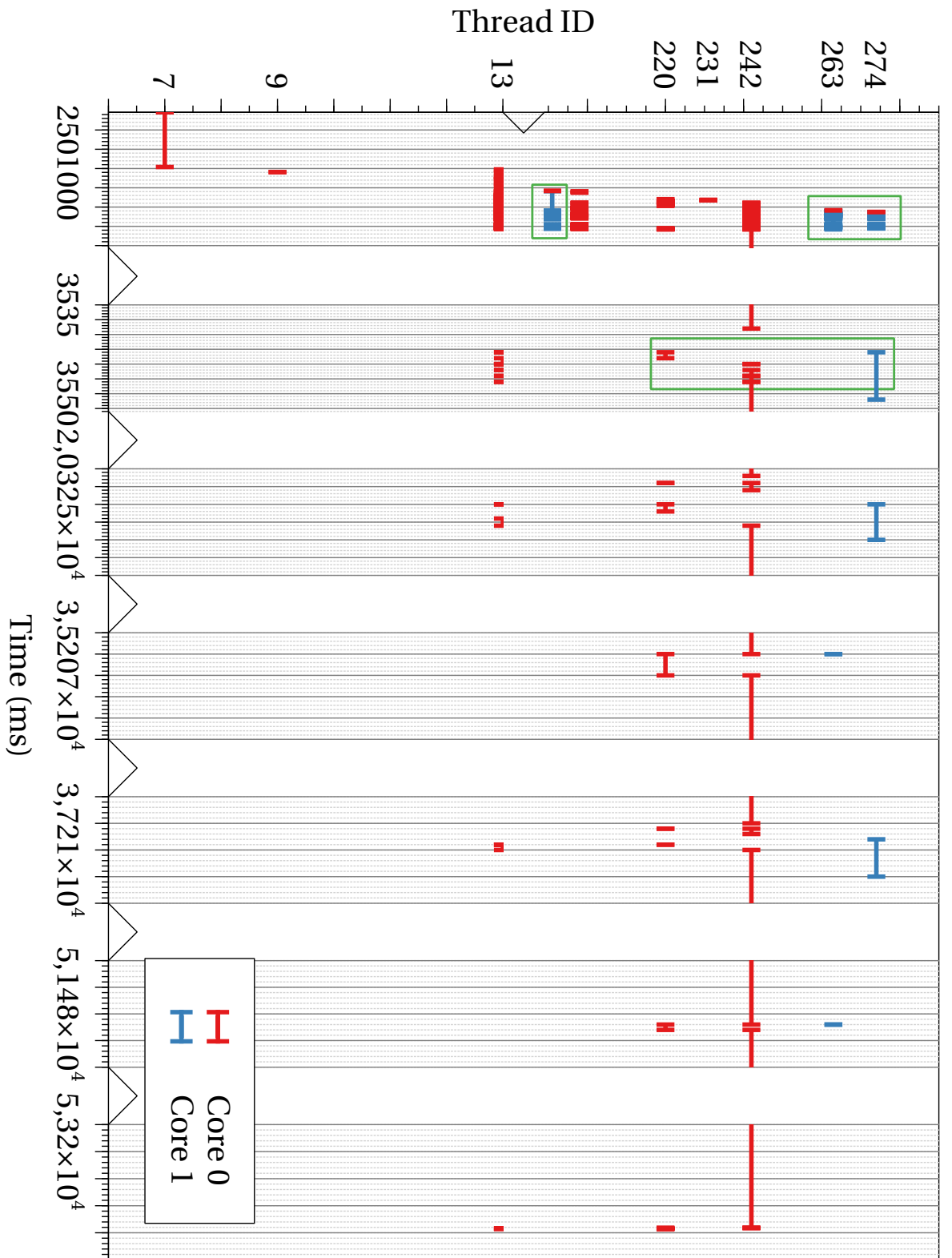


Figure 7.5: Plot shows a periodically executed task on *Core1* using a timing service on *Core0*

7.2 Scenario II: Testing System Properties using Artificially Generated Task Sets

By disabling the usage of services from components executed on *Core0*, components are no longer influencing the behavior of *Core0* during their actual execution on *Core1*. For this test, a long running task which calculates the fractional portion of the number PI will be used. The actual calculation doesn't depend on any services provided by the components on *Core0*. Plot 7.6 shows the result of this test. Again, bars colored in blue represent the timing behavior of *Core1*, whereas bars in red are indicating the behavior of *Core0*. The numbers which are labeled along the y-axis represent the process ids executed on each core:

- pi: 139, 146
- entry-point: 173, 84, 13
- signal: 184, 95
- system: 7

Similar to the first test, pi's main thread (139), entry-point thread (173) and signal thread (184) are create on *Core0* and will be moved to *Core1*. This is also marked with the two green boxes on the left-hand side of the plot. The green box located in the middle of plot marks the time-span in which the calculation of PI's fractional portion takes place. Within this area, there is actually no computation on *Core0* (i.e. idle). After the calculation is done, there is a interchange of information via signaling. This indicates that the remaining interplay is restricted to common management functionality. The green box on the right-hand side indicates that even independent workloads are possible.

So far, an interplay is still existent. It is possible, however, that a task on *Core1* can be executed without any interference caused by components on *Core0*. The key are the services which are used by a component and, maybe more important, where these services are located. In general, there are two possibilities for the usage of services. First, a component completely avoids the explicit usage of services for its own computation. This could be a solution in cases where components are not related to any input/output operations. Second, required services are bundled with the application on the same core. This solution however depends on the modularity of the used system where some components may fail if their dependent services are located on a different core. Nevertheless, the second approach allows the design of a service hierarchy where frequently used services can be located on the application core. Management services however can be assigned to a central core. In this way an interplay between components can be controlled and deferred. In the previous tests, *Core0* embodies such a simple management core which is used to get the application task (e.g. idle) up and running on *Core1*. In further tests, the components of the flexible task management were therefore added exclusively to *Core0*. Plot 7.7 shows this case where, beside the basic Genode components, the following components are executed on *Core0*:

- pi(1252)

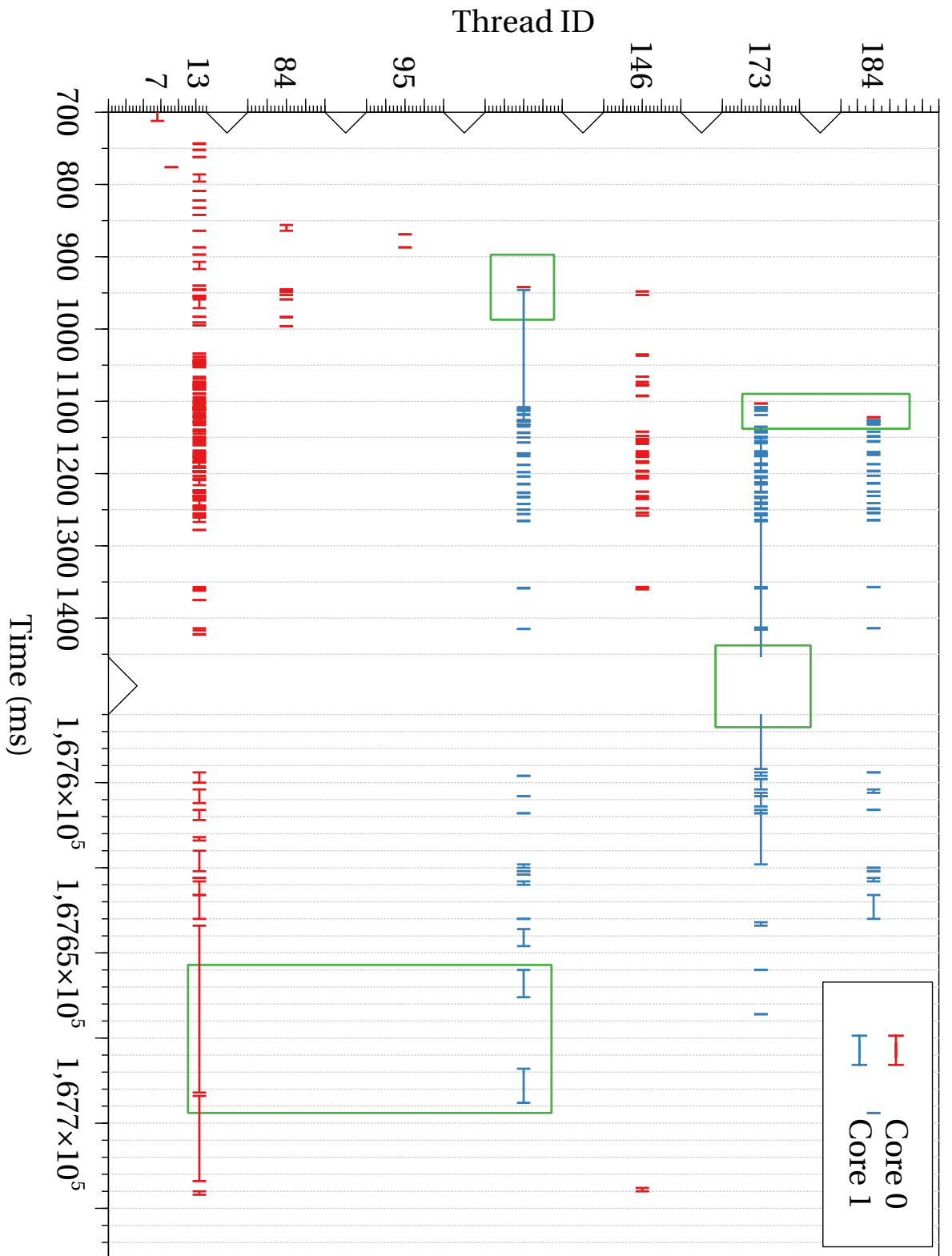


Figure 7.6: Plot shows a task calculating PI on *Core1* without using any services on *Core0*

- dom0-server(1224,298)
- nic(1214,563)
- sync(511)
- sched-controller(458)
- parser(405)
- taskloader(351)
- utilization(245)
- mon-manager(193)

These component modules are used as described in chapter 6 and are responsible for the deployment of new tasks which are received over the network. On the second core, *Core1*, pi(1299), signal(1334), entry-point(1323) are executed as before. The components are used to get task on *Core1* up and running. From the point in time where the actual calculation of pi takes place, there is no interruption from *Core0* and vice versa. The usage of the developed management components therefore doesn't cause an additional dependency.

In conclusion, this section has outlined the basic possibilities of separating the management components from the actual application components. To enable a separation it is mandatory to clarify which dependencies between components executed on distinct cores can be arise. The several tests have shown that there are principal interactions depending on a component's service usage. This usage however can be avoided to a certain degree which allows an independent workload for each core. According to this, flexible task management components were assigned to *Core0*. These tests are independent from a coexistent scheduling approach and were therefore done by using a fixed priority preemptive scheduling on both cores. The next section will cover the evaluation of an overload avoidance support by using a coexistent scheduling approach.

7.2.2 Support for the Avoidance of Overload Situations

This section will demonstrate the support for the avoidance of overload situations which will occur by an increasing workload on the system. As already known, an earliest deadline first scheduler can benefit from this support. The optimizing process is therefore executed for a set of earliest deadline first tasks. The test foresees that new tasks are periodically added to the system to drive the system to an overload situation. The optimizing mechanism shall circumvent this situation. Moreover, it is relevant how long the optimizing process takes place. Thus, a first timing measurement of the overall optimizing process will be given in the end.

The test configuration is based on a co-existent scheduling assignment (i.e. each core is managed by an other scheduling strategy). In case of the used dual-core system: *Core0* is managed by a fixed priority preemptive scheduling strategy. *Core1* is managed

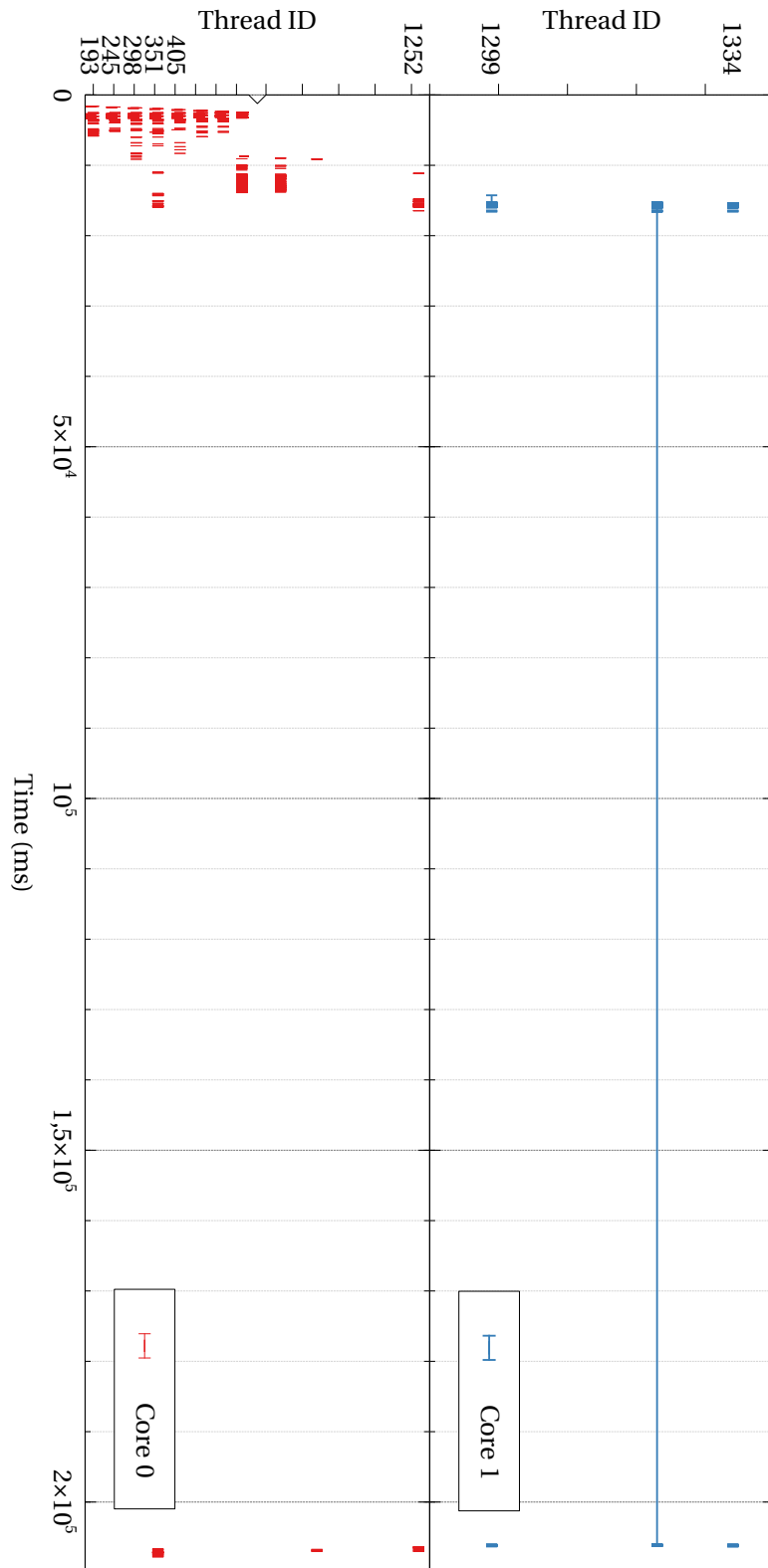


Figure 7.7: Plot shows a task calculating PI on Core1 with management components running on Core0

7.2 Scenario II: Testing System Properties using Artificially Generated Task Sets

by an earliest deadline first scheduling strategy. The used optimizer (i.e. *Sched_opt*) is used to optimize the tasks on *Core1* and uses the *utilization* as its decision function. Plot 7.8 shows the overall run of *Core1*. In general, it can be seen that the system constantly executes between two or three tasks simultaneously although the number of tasks increase over time. Tests without the optimizer, in contrast, showed that the system is getting overloaded with a task set between four and five tasks. With an active optimization the number of deployed tasks can be doubled. In result, there are eight tasks which can be executed on the core. The graph depicted in the bottom third of the plot indicates the distribution of tasks which could not be scheduled at a certain point in time. Additionally, the dots in the middle of the plot shows the point in time where a task was not allowed to be scheduled. In principal, the optimizer drops each task which causes a deadline miss. Of course this extreme optimization can lead to situations where six tasks are not allowed to be scheduled. The system, however, stays in a responsive state.

Optimizing processes can be very expensive and especially in the context of embedded systems, an adequate timing under the given resource constraints can be a challenge. The following timings could be measured during a single optimization step:

- Minimum: 27 ms
- Average: 109 ms
- Maximum: 376 ms

In relation to the remaining steps which are required for the deployment of a task via the network, these timings are only a fraction of which can be seen in figure 7.9. Over half of the required time is related to the network, namely sending the task description and the task binaries.

To sum it up, this section demonstrates that a potential overload situation can be avoided by the realized optimizing mechanism. Some situations however prevent scheduling up to six tasks. Though this guarantees a robust system execution but can be improved. Considering the required time of a single optimization step, it can be seen that even in the context of an embedded system this can be in an acceptable dimension, especially in comparison to the other steps which are required to deploy a task on the system. With this section, also the co-existent scheduling approach can be successfully demonstrated. The timing measurements are exclusively done for the optimizing step. In the next section thus the timing analysis for a acceptance test will be provided.

7.2.3 Support for Update Induced Dynamic Task Set Changes

For the test results provided in this section, each core of the dual-core system are managed by a fixed priority preemptive scheduling strategy. Due to the fact that tasks with a period will be deployed within this setup, the controller automatically chooses the acceptance test rather than the optimization method. The acceptance test is hereby realized as a combination of the response-time analysis and a sufficient test. The timings

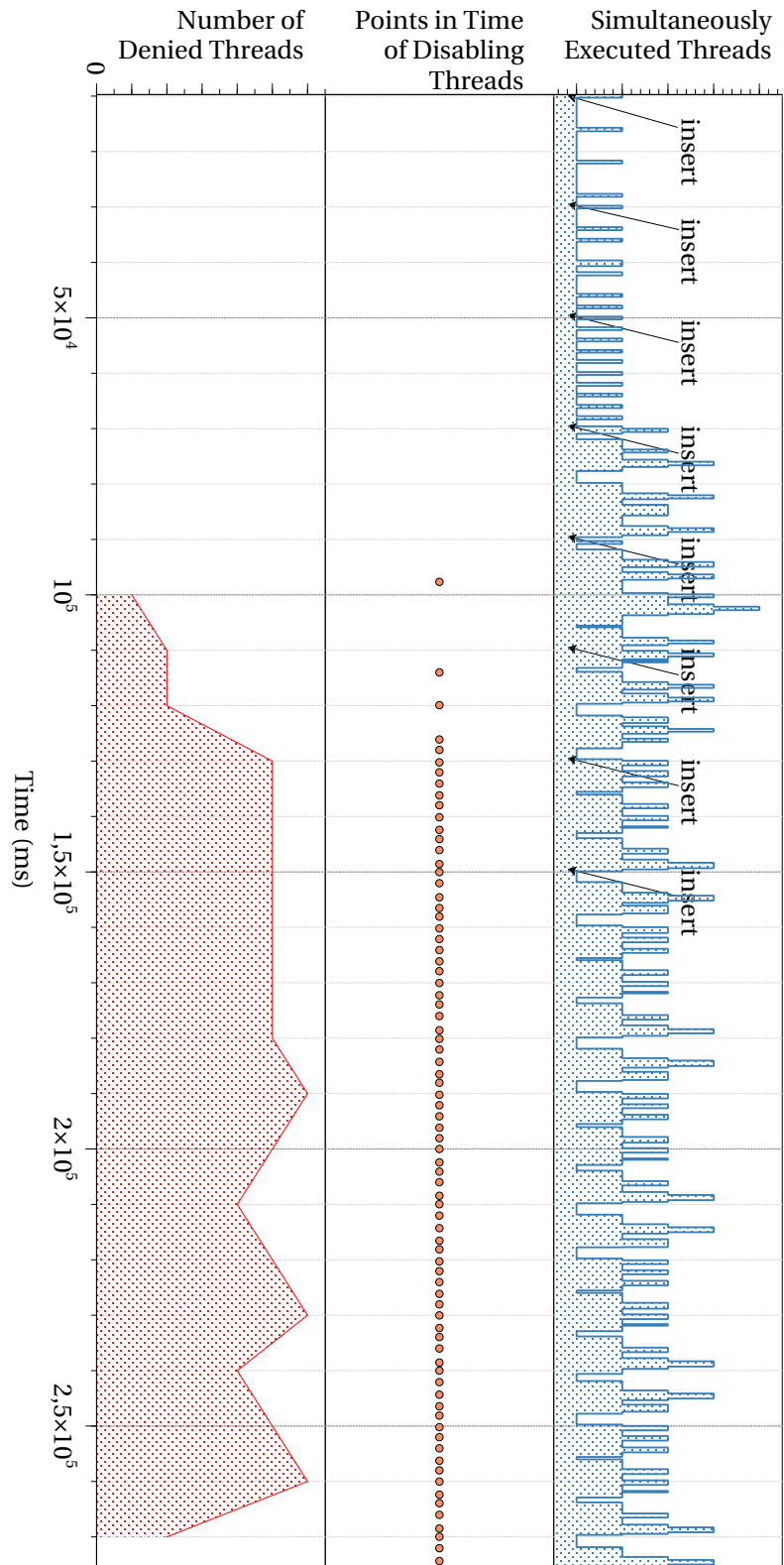


Figure 7.8: Histogram View of optimizing process

7.2 Scenario II: Testing System Properties using Artificially Generated Task Sets

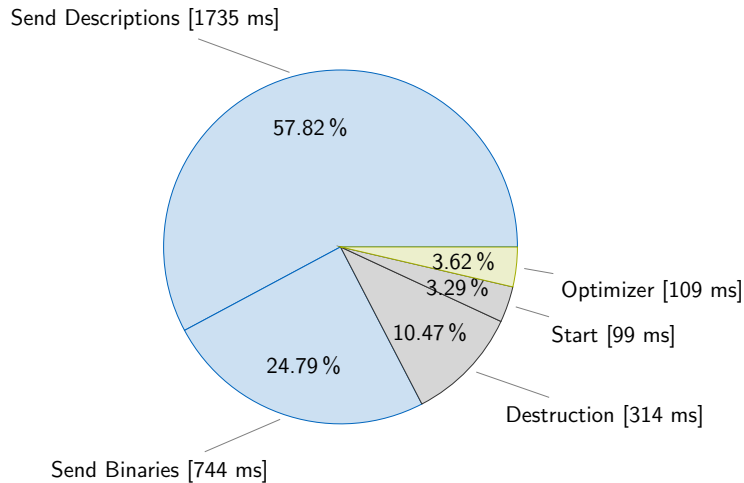


Figure 7.9: Piechart of average timing demands of several execution steps

for an acceptance test with a response-time analysis in contrast to the remaining steps needed for the deployment of a task can be seen in figure 7.10. Again, the time required for the analysis itself (*RTA Analysis*) is small in comparison to the other steps. The steps for sending the task description and the task binaries require the majority of the time. These are single steps of one deployment process.

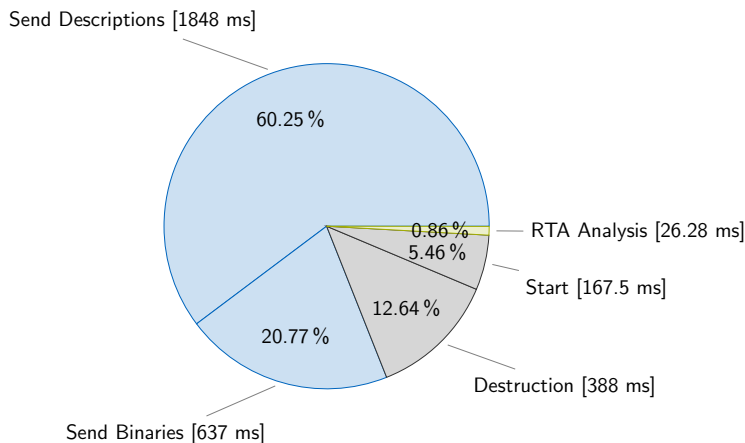


Figure 7.10: Piechart of average timing demands of several steps during execution

The figure 7.11 presents the timings of certain deployment processes, all for different applications. The measurement starts after the system connects to the workstation and is terminated after the first acknowledgment of the system (i.e. exit event if task terminates). The dark blue bar *time since desc* shows the duration it takes from the first connect until the description was send. In contrast, the light blue bar (*push profile*) shows the duration of the overall deployment process on the system. It describes the

7 Evaluation of the Flexible Task Management

time between the start of the task and the response of the system to the workstation. Again, the actual time spent for the execution of the deployment of a given task into the system is by far smaller than it takes to send the description.

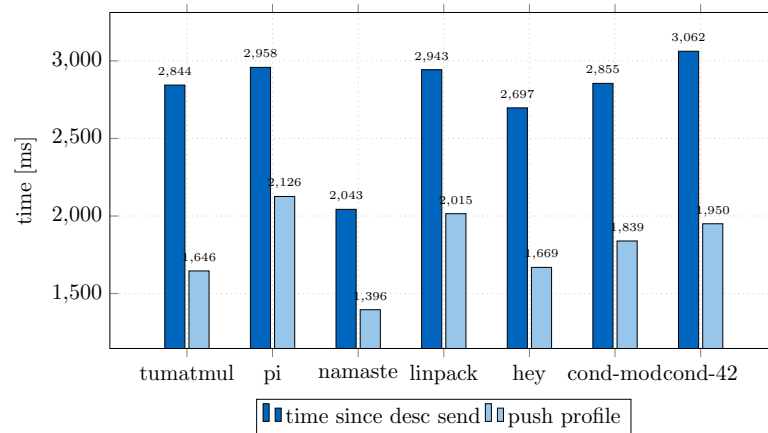


Figure 7.11: Timing measurements of different applications

Within this section more timing related measurements were given. The focus hereby relies on the acceptance test rather than the optimizer. Furthermore, the overall timing of a single deployment process as well as a number of deployment processes were provided. It can be seen, that the network related steps always require a great amount of time of the deployment process. This section closes the evaluation tests considering an artificial task set. In the following section, the evaluation results should provide a possible usage of the invented approach in an automotive environment.

Conclusion

This chapter has described first evaluation results of the realized flexible task management approach. It therefore has provided two use-case scenarios in where the operating system is executed. Several contributions regarding isolation, timing and optimizing where done in the standalone scenario. A principal applicability in an automotive context was provided by the autonomous driving scenario. Following this, the next chapter will provide an overall conclusion about the thesis and will point out possible future improvements.

8 Conclusion

This chapter will describe the relations between the research questions, contributions, and the respective chapters in summary. The focus here relies on answering the initial research questions considering the contributions and the thesis chapters. The ordering within this chapter conforms to the ordering of the research questions. As a result, final remarks will be given in this chapter. Open points for future development however will be subject to chapter 9.

Resource Partitioning

Automotive trends lead to a consolidation of hardware units which may result in systems where software with different criticality classifications are executed. An important aspect of such mixed-critical systems is the partitioning of available resources between the software. On the other side, a flexible management of resource is needed for future connected cars. A partitioning approach needs to be found which is capable of separating both software with different criticality levels and the management from the actual execution.

The approach which is contributed (cp. contribution 1c) by this work in general foresees the usage of a multi-core platform where each core is managed by a separate scheduling strategy (i.e. co-existent scheduling).

Research Question 1 The separation between applications and adaptation management was addressed by the proposed architecture design (cp. contribution 2a) in chapter 4. A possible allocation of software components to the multi-core hardware platform considering this separation was discussed in section 4.3.1. The provided evaluation results (cp. contribution 5a) of both automotive scenarios in section 7.1.1 and section 7.1.2 have demonstrated the successful partitioning of applications and adaptation management.

Research Question 2 The critical aware allocation of an application during run-time was again addressed by the proposed architecture design (cp. contribution 2a) in chapter 4. A concept for the critical-aware allocation of applications was described in section 4.3.2. With this concept, a new dispatching mechanism as part of an integration framework (cp. contribution 3c) was designed in section 5.2.1. The evaluation results in chapter 7 have demonstrated, that an allocation according to the criticality of an application can be used to assign it to a corresponding scheduler.

In general, The final tests in chapter 7 shows that the architecture and the implementation are able to handle the separation and that the usage of co-existent scheduling strategies is feasible. A complete separation, however, has not been achieved so far due to certain reasons. A main concern are the services which are provided by software components on one core and are used by software components (i.e. application) of another core. The dependencies between the components, indeed, can be reduced but not avoided. A management of cores by co-existent scheduling strategies however can be proofed without restrictions. If a scheduling strategy corresponds to a set of tasks with a certain criticality classification, even the execution of mixed-critical systems is possible. But, due to the dependencies between the cores, a restriction is that the cores have to be the same criticality. As an alternative, a high-critical core is allowed to manage a low-critical core. These variants were successfully tested in chapter 7.

Adaptation Control

Following the trend of hardware consolidation, the integration of a flexible task management along the actual application on the same device is reasonable. This thesis therefore contributes to a concept which allows such a system configuration. Related to the context of mixed-critical systems, an unconditional system adaptation through the flexible task management needs to be avoided. Controlling the adaptation and guaranteeing certain aspects about the system's behavior is therefore a matter of interest.

Research Question 3 Integrating an adaptation control on top of an operating system was considered in the overall architecture design which was presented in section 4.1 and section 4.2 following the contribution 2b. At first, relevant software components forming the flexible task management are identified and designed. Additionally, chapter 4 evaluates several design possibilities how the main components of the flexible task management can be integrated in the system on the same device. Moreover, it was important to find a representation of the underlying kernel space and an extended application model (cf. contribution 3b) which was presented in section 5.1. Both aspects were considered in the later implementation of the software components (cf. contribution 4b).

Research Question 4 For designing a concrete adaptation process which supports dynamic mixed-criticality workloads, concrete workflows for the adaptation process were identified (cf. contribution 3a) in section 4.4. These workflows are used to identify potential data and control flows to clarify in which operational states a system could be (see chapter 4). This information is used to design an adaptation process which can be done during the operation of a system (see chapter 5). The actual adaptation process is based on a run-time integration framework (cf. contribution 3c) which combines the dispatching, admission and synchronization steps in one consistent process. This framework was described in section 5.2. The concrete mechanisms were identified and realized in chapter 5 and chapter 6 respectively. Two case studies were provided in chapter 7 which show the the basic functionality of the implemented control mechanism by using an automotive installation and update scenario.

Research Question 5 A concrete control mechanism that is able to keep a robust system execution was developed as part of the run-time integration framework (cf. 3c). In the concrete case an optimization process for avoiding overload situations was developed in section 5.2.3. The possible self-optimization (cf. contribution 5c) was successfully evaluated in section 7.2.2.

Finally, within the thesis relevant software components for realizing a flexible task management could be identified. With the realization of the task management, an adaptation during system's operation can be demonstrated. The flexible task management and the applications can be integrated on the same device whereas the realization is following the outlined architecture approach. In combination with restrictions in the separation of components, the thesis proposes a central management core where the flexible task management is executed. Additionally, chapter 6 already provides some limitations of the current implementation.

Resource Usage and Overhead

In the area of embedded systems, a provided solution needs to deal with resource constraints. Related to this context, this thesis contributes to the investigation of efficient adaptation mechanisms (i.e. low overhead and resource consumption) applicable to this area.

Research Question 6 Investigating in the overall timing behavior of the provided adaptation process (cf. contribution 5d), the evaluation results that were presented in section 7.2.3 has proven that the proposed approach introduces an acceptable timing overhead in contrast to other stages of the adaptation process (see chapter 7). Moreover, the core decision making provided by this approach is rather efficient even in the context of an embedded system. The overall contributions show, that an application of self-x capable decision making algorithms directly on the embedded device is feasible. Of course, there is an overhead but it is minimal in comparison to other steps of the deployment process. Moreover, guarantees of a reliable system execution can be shown by using the developed self-optimizing approach. Although the first tests are promising, the timing requirements heavily depend on the number of to be tested tasks.

To sum it up, this chapter has concluded the research question and the contributions made by this thesis. For each contribution, related chapters were mentioned and final remarks were given. There are still open aspects which will be covered in the next chapter.

9 Future Work

The final remarks given in chapter 8 provide a starting point for a future development of the proposed solution. There are several directions in which the system can be evolved.

Resource Partitioning In general, the main focus of this thesis relies on the management of cores (i.e. scheduling). A resource partitioning however also needs to consider the available memory resources of a system. A possible extension of the developed approach is to modify the outlined adaptation management in this direction. In the concrete case, the information gathering and the decision making process need to be extended. Related to this, a partial separation is provided within this thesis. Investigating in hardware architectures with multi-processor configuration could solve the shared memory (i.e. caches) problem. On the other side, transforming the micro-kernel based approach to an exo-kernel allows to decouple the components further. The deployed applications are meant to be independent from other tasks to provide their calculations. In an automotive system however several functions are bundled together to realize a distinct application. The communication and synchronization between these tasks needs to be investigated. In addition, the provided approach lacks a safety certification which would be required for the application in future vehicles.

Adaptation Control Considering the current implementation of the adaptation control mechanism, there are several open points for future work. Like the developed controlling component, the synchronization component needs to be enhanced by adequate algorithms to realize an intelligent decision, when user-space and kernel-space can be synchronized. In general, the adaptation of the kernel-space from the user-space embodies some challenges. In the long run, an user-space scheduling could be feasible where the micro-kernel is then solely responsible for messaging and capability management. This would reduce the complexity further and allows the reinforcement as a trusted computing base. As a central point hereby would be an efficient representation of a kernel-space state in user-space. The so far provided algorithms for realizing a self-optimizing system are a first step in this direction. Further paradigms and algorithms however need to be applied.

Resource Usage and Overhead By getting more and more powerful embedded systems, new opportunities for the deployment and management of applications arise. Applications used in this thesis contained one single threaded task. An extension however could be to deploy container-like compartments which host complete systems like Android or Linux. The usage scenarios would be similar to data-centers. Furthermore, more powerful co-processor configuration and thus algorithms could be developed for this class of

9 Future Work

devices. This leads to a possible extension where the management core is a co-processor or the decision making algorithms are bound to a single special-purpose processor. In general, the provided architecture approach could be enhanced to utilize such hardware configurations leading to a further reduction of potential overhead.

A Code Listings

A.1 Source Code

```
1 void Platform_thread::_finalize_construction(const char *name)
2 {
3     if (_dl>0)
4     {
5         _prio=0;
6         params = l4_sched_param_by_type(Deadline, _dl, 0);
7     }
8     else if(_prio>0)
9     {
10        params = l4_sched_param_by_type(Fixed_prio, _prio, 0);
11    }
12    else {
13        PWRN("wrong_scheduling_type_prio:%d_deadline:%d", _prio, _dl);
14        return;
15    }
16
17    l4_scheduler_run_thread(L4_BASE_SCHEDULER_CAP, _thread.local.data ←
18        ()->kcap(),
19        &params);
20 }
```

Listing A.1: example of using the created scheduling context in user-space

```
1 l4_scheduler_run_thread_u(l4_cap_idx_t scheduler, l4_cap_idx_t ←
2     thread,
3     l4_sched_param_t const *sp, l4_utcb_t *utcb) L4_NOTHROW
4 {
5     l4_msg_regs_t *m = l4_utcb_mr_u(utcb);
6     m->mr[0] = L4_SCHEDULER_RUN_THREAD_OP;
7     m->mr[1] = (sp->affinity.granularity << 24) | sp->affinity.offset ←
8     ;
9     m->mr[2] = sp->affinity.map;
10    m->mr[3] = sp->prio;
11    m->mr[4] = sp->quantum;
12    m->mr[5] = sp->deadline; /* Own work */
13    m->mr[6] = l4_map_obj_control(0, 0);
14    m->mr[7] = l4_obj_fpage(thread, 0, L4_FPAGE_RWX).raw;
15    return l4_ipc_call(scheduler, utcb, l4_msgtag(L4_PROTO_SCHEDULER, ←
16        6, 1, 0), L4_IPC_NEVER);
17 }
```

14 }

Listing A.2: scheduler interface for running a thread

```

1 Scheduler::sys_run(L4_fpage::Rights, Syscall_frame *f, Utcb const * iutcb, Utcb *outcb)
2 {
3     /* edf thread */
4     if (iutcb->values[5] > 0)
5     {
6         L4_sched_param_deadline sched_p;
7         sched_p.sched_class = -3;
8         /* Add deadline to arrival time */
9         sched_p.deadline = (iutcb->values[5])+(outcb->values[1]);
10        thread->sched_context()->set(static_cast<L4_sched_param*>(& sched_p));
11        sched_param = reinterpret_cast<L4_sched_param const *>(&sched_p);
12        info.sp=sched_param;
13    }
14    else
15    {
16        /* fp thread */
17        if (iutcb->values[3] > 0)
18        {
19            L4_sched_param_fixed_prio sched_p;
20            sched_p.sched_class = -1;
21            sched_p.prio = iutcb->values[3];
22            thread->sched_context()->set(static_cast<L4_sched_param*>(& sched_p));
23            sched_param = reinterpret_cast<L4_sched_param const *>(&sched_p);
24            info.sp=sched_param;
25        }
26    }

```

Listing A.3: run function of scheduler in Fiasco.OC

```

1 Sched_context::set(L4_sched_param const *_p)
2 {
3     switch (p->p.sched_class)
4     {
5
6         case L4_sched_param_deadline::Class:
7             if (p->deadline.deadline == 0)
8                 return -L4_err::EInval;
9             _t = Deadline;
10            _sc.edf._p = 0;
11            _sc.edf._dl = p->deadline.deadline;
12            _sc.edf._q = Config::Default_time_slice;
13

```

```

14     break;
15
16     case L4_sched_param_fixed_prio::Class:
17         _t = Fixed_prio;
18
19         _sc.fp._p = p->fixed_prio.prio;
20         if (p->fixed_prio.prio > 255)
21             _sc.fp._p = 255;
22
23         if (p->fixed_prio.quantum == 0)
24             _sc.fp._q = Config::Default_time_slice;
25         else
26             _sc.fp._q = p->fixed_prio.quantum;
27
28         break;
29
30     default:
31         return L4_err::ERange;
32 };
33 return 0;
34 }

```

Listing A.4: set function in scheduling context

```

1 L4_INLINE l4_msgtag_t
2 l4_thread_stats_time(l4_cap_idx_t thread) L4NOTHROW
3 {
4     return l4_thread_stats_time_u(thread, l4_utcb());
5 }

```

Listing A.5: Entry point for user-space interface in Fiasco.OC

```

1 L4_INLINE l4_msgtag_t
2 l4_thread_stats_time_u(l4_cap_idx_t thread, l4_utcb_t *utcb) ↵
3     L4NOTHROW
4 {
5     l4_msg_regs_t *v = l4_utcb_mr_u(utcb);
6     v->mr[0] = L4_THREAD_STATS.OP;
7     return l4_ipc_call(thread, utcb, l4_msgtag(L4_PROTO_THREAD, 1, 0, ↵
8         0), L4_IPC_NEVER);
9 }

```

Listing A.6: The internal thread_stats_time function

```

1 Thread_object::invoke(L4_obj_ref /*self*/, L4_fpage::Rights rights, ↵
2     Syscall_frame *f, Utcb *utcb)
3 {
4     switch (utcb->values[0] & Opcode_mask)
5     {

```

A Code Listings

```
5     case Op_stats:
6         f->tag(sys_thread_stats(f->tag(), utcb));
7         return;
8     }
9 }
```

Listing A.7: invoke method of Thread_object

```
1 Thread_object::sys_thread_stats(L4_msg_tag const & /*tag*/, Utcb * ←
    utcb)
2 {
3     Clock::Time value;
4     Clock::Time _period;
5     Clock::Time _start;
6     Clock::Time _dead;
7     Clock::Time _time;
8
9     if (home_cpu() != current_cpu())
10        drq(handle_sys_thread_stats_remote, &value, Drq::Any_ctxt);
11    else
12        {
13        // Respect the fact that the consumed time is only updated on ←
            context switch
14        if (this == current())
15            update_consumed_time();
16        value = consumed_time();
17        _period = period();
18        _start = start_time();
19        _dead = get_dead_time();
20        _time = get_time();
21        }
22
23    utcb->values[0] = value;
24    utcb->values[1] = _period;
25    utcb->values[2] = _start;
26    utcb->values[3] = _dead;
27    utcb->values[4] = _time;
28    //printf("%lu %lu %lu %lu %lu\n", utcb->values[0], utcb->values[1], ←
        utcb->values[2], utcb->values[3], utcb->values[4]);
29
30    return commit_result(0, 5); // Utcb::Time_val::Words);
31 }
```

Listing A.8: central function to get timing-related information about a thread

```
1 class Context :
2     public Context_base,
3     protected Rcu_item
4 {
5 public:
6     Cpu_time consumed_time();
```



```

7   Cpu_time start_time();
8   void set_dead_time(Clock::Time dead);
9   Cpu_time get_dead_time();
10  Cpu_time get_time();
11 };
12 Unsigned64
13 Context::period() const
14 {
15     return _period;
16 }

```

Listing A.9: Base class of scheduling context with relevant timing attributes

```

1  L4_INLINE l4_msgtag_t
2  l4_scheduler_deploy_thread(l4_cap_idx_t scheduler,
3  int* thread) L4NOTHROW
4  {
5  return l4_scheduler_deploy_thread_u(scheduler, thread, l4_utcb()) ←
6  ;
7  }

```

Listing A.10: interface function called from user-space component

```

1  enum L4_scheduler_ops
2  {
3  L4_SCHEDULER_INFO_OP      = 0UL, /**< Query infos about the ←
4  scheduler */
5  L4_SCHEDULER_RUN_THREAD_OP = 1UL, /**< Run a thread on this ←
6  scheduler */
7  L4_SCHEDULER_IDLE_TIME_OP  = 2UL, /**< Query idle time for the ←
8  scheduler */
9  L4_SCHEDULER_DEPLOY_THREAD_OP = 3UL, /**< Query idle time for ←
10 the scheduler */
11 L4_GET_RQS = 4UL,
12 L4_GET_DEAD = 5UL,
13 };

```

Listing A.11: Additional operation for deploying threads

```

1  L4_INLINE l4_msgtag_t
2  l4_scheduler_deploy_thread_u(l4_cap_idx_t scheduler, int* thread, ←
3  l4_utcb_t *utcb) L4NOTHROW
4  {
5  l4_msg_regs_t *m = l4_utcb_mr_u(utcb);
6  m->mr[0] = L4_SCHEDULER_DEPLOY_THREAD_OP;
7  m->mr[1] = thread[0];
8  m->mr[2] = thread[1];
9  for(int i = 1; i < thread[0]+1; i++){
10     m->mr[2*i+1] = thread[2*i];

```

A Code Listings

```
10     m->mr[2*i+2] = thread[2*i+1];
11     }
12     return l4_ipc_call(scheduler, utcb, l4_msgtag(L4_PROTO_SCHEDULER, ←
        (2*thread[0])+3, 1, 0), L4_IPC_NEVER);
13 }
```

Listing A.12: internal interface function for thread deployment

```
1 class Scheduler : public Icu_h<Scheduler>, public Irq_chip_soft
2 {
3 public:
4     enum Operation
5     {
6         Info          = 0,
7         Run_thread   = 1,
8         Idle_time    = 2,
9         Deploy_thread = 3,
10        Get_rqs       = 4,
11        Get_dead      = 5,
12    };
13 };
```

Listing A.13: Internal options within scheduler object

```
1 PUBLIC
2 L4_msg_tag
3 Scheduler::kinvoke(L4_obj_ref ref, L4_fpage::Rights rights, ←
    Syscall_frame *f,
4     Utc const *iutcb, Utc *outcb)
5 {
6     switch (iutcb->values[0])
7     {
8     case Info:          return sys_info(rights, f, iutcb, outcb);
9     case Run_thread:   return sys_run(rights, f, iutcb, outcb);
10    case Idle_time:    return sys_idle_time(rights, f, outcb);
11    case Deploy_thread: return sys_deploy_thread(rights, f, iutcb);
12    case Get_rqs:      return sys_get_rqs(rights, f, iutcb, outcb);
13    case Get_dead:    return sys_get_dead(rights, f, outcb);
14    default:          return commit_result(-L4_err::ENosys);
15    }
16 }
```

Listing A.14: kinvoke method with switch statement for different operations

```
1 PRIVATE
2 L4_msg_tag
3 Scheduler::sys_deploy_thread(L4_fpage::Rights, Syscall_frame *f, ←
    Utc const *utcb) //gmc
4 {
```

```

5  L4_msg_tag const tag = f->tag();
6
7  Sched_context::Ready_queue &rq = Sched_context::rq.current();
8
9  int list[(int)tag.words()-1];
10 list[0]=((int)tag.words()-3)/2;
11
12 for(int i = 1 ; i < tag.words()-1; i++){
13     list[i]=utcb->values[i+1];
14 }
15
16 rq.switch_ready_queue(&list[0]);
17
18 return commit_result(0);
19 }

```

Listing A.15: Deploy thread function within the scheduler object

```

1     bool switch_ready_queue(int* info) //gmc
2     {
3         return switch_rq(info);
4     }

```

Listing A.16: Wrapper function for the actual switch_rq function within Ready_queue_base class

```

1 IMPLEMENT
2 bool
3 Sched_context::Ready_queue_base::switch_rq(int* info) //gmc
4 {
5     if(info[1]==0)
6         return fp_rq.switch_rq(info);
7     else
8         return edf_rq.switch_rq(info);
9 }

```

Listing A.17: Actual switch_rq functio in Ready_queue_base

```

1 class Ready_queue_fp
2 {
3 public:
4     bool switch_rq(int* info) {
5         for(int i=0;i<ordered;i++)
6         {
7             //printf("enqueue %d\n", Kobject_dbg::obj_to_id(ordered_contexts ←
8                 [i]));
9             requeue(ordered_contexts[i]);
10        }
11        return true;

```

A Code Listings

```
11     }
```

Listing A.18: Concrete implementation of `switch_rq` by reusing Fiasco.OCs internal queuing methods

```
1  int Dom0_server::connect()
2  {
3      socklen_t len = sizeof(_target_addr);
4      _target_socket = lwip_accept(_listen_socket, &_amp;_target_addr, &len) ←
5      ;
6      if (_target_socket < 0)
7      {
8          PWRN("Invalid _socket _from _accept!");
9          return _target_socket;
10     }
11     = sockaddr_in* target_in_addr = (sockaddr_in*)&_target_addr;
12     PINF("Got _connection _from _%s", inet_ntoa(target_in_addr));
13     return _target_socket;
14 }
```

Listing A.19: `genode-dom0-hw/server.cc` `connect`

```
1  void Dom0_server::serve()
2  {
3      while (true)
4      {
5          if (message == SEND_DESCS) {...}
6          else if (message == SEND_BINARIES) {...}
7          else {...}
8      }
9  }
```

Listing A.20: `genode-dom0-hw/server.cc`

```
1  struct Monitoring_object
2  {
3      Genode::Session_label  session_label;
4      Genode::Trace::Thread_name  thread_name;
5      Genode::Trace::Policy_id  policy_id;
6      Genode::Trace::Execution_time  execution_time;
7      unsigned                prio;
8      unsigned                id;
9      unsigned                foc_id;
10     size_t                  ram_quota;
11     size_t                  ram_used;
12     Genode::Trace::CPU_info::State  state;
13     Genode::Affinity::Location  affinity;
14     unsigned long long      start_time;
15     unsigned long long      arrival_time;
```

```

16 unsigned long long exit_time;
17 };

```

Listing A.21: central element of information gathering - the monitoring object

```

1 size_t Mon_manager::update_info(Genode::Dataspace_capability ←
    mon_ds_cap)
2 {
3   num_threads=100;
4   Monitoring_object *threads = Genode::env()->rm_session()->attach( ←
    mon_ds_cap);
5   static Genode::Trace::Connection trace(1024*4096, 64*4096, 0);
6   Genode::Trace::Subject_id subjects[num_threads];
7   size_t num_subjects = trace.subjects(subjects, num_threads);
8   for (size_t i = 0; i < num_subjects; i++) {
9     Genode::Trace::CPU_info cpu_info = trace.cpu_info(subjects[i]);
10    Genode::Trace::RAM_info ram_info = trace.ram_info(subjects[i]);
11    threads[i].session_label=ram_info.session_label();
12    threads[i].thread_name=ram_info.thread_name();
13    threads[i].prio=cpu_info.prio();
14    threads[i].execution_time=cpu_info.execution_time();
15    threads[i].id=cpu_info.id();
16    threads[i].foc_id=cpu_info.foc_id();
17    threads[i].ram_used=ram_info.ram_used();
18  }
19  return num_subjects;
20 }

```

Listing A.22: main routine of mon.manager for getting system information (shortened)

```

1 CPU_info Session-component::cpu_info(Subject_id subject_id)
2 {
3   return _subjects.lookup_by_id(subject_id)->info_cpu();
4 }
5 }

```

Listing A.23: Implementation of cpu_info

```

1 class Genode::Trace::Subject
2 {
3 public:
4   CPU_info info_cpu()
5   {
6     Execution_time execution_time;
7     {
8       if (source.valid()) {
9         Trace::Source::Info const info = source->info();
10        execution_time = info.execution_time;
11      }

```

A Code Listings

```
12     }
13     CPU_info info= CPU_info(_state(), _policy_id ,
14                          execution_time , affinity , start_time , ←
                              arrival_time , kill_time , prio , id , foc_id ←
                              , pos_rq
15     );
16     return info;
17 }
18 };
```

Listing A.24: Implementation of info_cpu

```
1 class Genode::Trace::Source
2 {
3     public:
4         Info const info() const
5         {
6
7             Info info;
8             Dynamic_Info dyn=_info.dynamic_info();
9             Static_Info stat=_info.static_info();
10            info.execution_time=dyn.execution_time;
11            return info;
12        }
13 };
```

Listing A.25: Implementation of basic info function

```
1 class Genode::Cpu_thread_component : public Rpc_object<Cpu_thread>,
2                                     public List< ←
                                         Cpu_thread_component >:: ←
                                         Element ,
3                                     public Trace::Source:: ←
                                         Info_accessor
4 {
5     private:
6         Platform_thread      _platform_thread;
7     public:
8         Trace::Source::Dynamic_Info dynamic_info() const
9         {
10            return { _platform_thread.execution_time() ,
11                   _platform_thread.affinity() ,
12                   _platform_thread.start_time() ,
13                   _platform_thread.arrival_time() ,
14                   _platform_thread.kill_time() ,
15                   _platform_thread.prio() };
16        }
17 };
```

Listing A.26: Implementation of dynamic.info function

```

1 Genode::Ram_dataspace_capability Parser_session_component:: ↵
   live_data()
2 {
3   mon_ds_cap = Genode::env()->ram_session()->alloc(100* sizeof( ↵
   Mon_manager:: Monitoring_object));
4   Mon_manager:: Monitoring_object *threads=Genode::env()->rm_session ↵
   ()->attach(mon_ds_cap);
5   dead_ds_cap = Genode::env()->ram_session()->alloc(256* sizeof(long ↵
   long unsigned));
6   long long unsigned *rip=Genode::env()->rm_session()->attach( ↵
   dead_ds_cap);
7   size_t num_subjects=_mon_manager.update_info(mon_ds_cap);
8
9   Genode::Xml_generator xml(_live_data.local_addr<char>(), ↵
   _live_data.size(), "live", [&]()
10  {
11    xml.node("task-descriptions", [&]()
12    {
13      for (int j = 0; j < 1; j++) {
14        for (size_t i = 0; i < num_subjects; i++) {
15          xml.node("task", [&]()
16          {
17            xml.attribute("id", std::to_string(threads[i].id). ↵
   c_str());
18            xml.attribute("foc_id", std::to_string(threads[i].foc_id). ↵
   c_str());
19            xml.attribute("execution-time", std::to_string(threads[i]. ↵
   execution_time.value/1000).c_str());
20            xml.attribute("priority", std::to_string(threads[i]. ↵
   prio).c_str());
21            xml.attribute("core", std::to_string(threads[i].affinity.xpos ↵
  ()).c_str());
22            xml.attribute("policy-id", std::to_string(threads[i]. ↵
   policy_id.id).c_str());
23            xml.attribute("state", std::to_string(threads[i].state).c_str ↵
  ());
24            xml.attribute("arrival-time", std::to_string(threads[i]. ↵
   arrival_time/1000).c_str());
25            xml.attribute("start-time", std::to_string(threads[i]. ↵
   start_time/1000).c_str());
26            xml.attribute("exit-time", std::to_string(threads[i]. ↵
   exit_time/1000).c_str());
27            xml.attribute("session", threads[i].session_label.string());
28            xml.attribute("thread", threads[i].thread_name.string());
29            xml.attribute("ram_quota", std::to_string(threads[i]. ↵
   ram_quota/1024).c_str());
30            xml.attribute("ram_used", std::to_string(threads[i].ram_used ↵
   /1024).c_str());
31          });
32        }
33        //Genode:: printf("run %d\n",j);
34        _mon_manager.update_info(mon_ds_cap);
35      }
36    });

```

A Code Listings

```
37  });
38  Genode::env()->ram_session()->free(mon_ds_cap);
39  return _live_data.cap();
40 }
```

Listing A.27: live_data function for serializing system information

```
1  int Sched_controller::enq(int core, Rq_task::Rq_task task)
2  {
3      if(task.task_class == Rq_task::Task_class::hi)
4      {
5          //Execute sufficient schedulability test
6          if (!fp_alg.fp_sufficient_test(&task, &_rqs[core]))
7          {
8              //If sufficient test fails —> execute RTA (exact test)
9              if (!fp_alg.RTA(&task, &_rqs[core]))
10             {
11                 return -1;
12             }
13         }
14     }
15     else if (task.task_class == Rq_task::Task_class::lo)
16     {
17         // do task optimization for lo tasks
18         _optimizer->add_task((unsigned int) core, task);
19     }
20     int success = _rqs[core].enq(task);
21
22     return success;
23 }
24 }
```

Listing A.28: adding a new task to the controllers ready queues

```
1  void Taskloader_session_component::add_tasks(Genode:: ←
2      Ram_dataspace_capability xml_ds_cap)
3  {
4      Genode::Region_map* rm = Genode::env()->rm_session();
5      const char* xml = rm->attach(xml_ds_cap);
6      if(verbose_debug) PINF("Parsing XML file:\n%s", xml);
7      Genode::Xml_node root(xml);
8      Rq_task::Rq_task rq_task;
9      //Update rq_buffer before adding tasks for online analyses to core ←
10     1
11     sched.update_rq_buffer(1);
12     const auto fn = [this, &rq_task] (const Genode::Xml_node& node)
13     {
14         _shared.tasks.emplace_back(_ep, _cap, _shared, node, &sched);
```



```

15 //Add task to Controller to perform a schedulability test for ←
    core 1
16 rq_task = _shared.tasks.back().getRqTask();
17 int result = sched.new_task(rq_task, 1);
18 if (result != 0){
19     if(verbose_debug) PINF("Task_with_id_%d_was_not_accepted_by_the_ ←
        controller", rq_task.task_id);
20     _shared.tasks.back().setSchedulable(false);
21 }
22 else{
23     if(verbose_debug) PINF("Task_with_id_%d_was_accepted_by_the_ ←
        controller", rq_task.task_id);
24     _shared.tasks.back().setSchedulable(true);
25 }
26 };
27
28 root.for_each_sub_node("periodictask", fn);
29
30 sched.set_opt_goal(xml_ds_cap);

```

Listing A.29: Adding a task is a simple xml parsing and deferred analysis

A.2 Accpetance Test in Pseudocode

Algorithm A.1 Response Time Analysis

```

if priority of new task is lower than all other tasks then
     $rtime_{old} \leftarrow ntask_{wcet}$ 
    if  $\neg$ CMP_RESPONSE_TIME(ntask, nelements, ntask) then
        return not schedulable
    end if
else compute response time for all tasks with  $t_{prio} \leq ntask_{prio}$ 
    for all  $t \in tasks$  do
        if  $t_{prio} \leq ntask_{prio}$  then
            if  $\neg$  CMP_RESPONSE_TIME(ntask,idx,t) then
                return not schedulable
            end if
        end if
        if  $t_{prio} \geq ntask_{prio}$  and  $(t+1)_{prio} \leq ntask_{prio}$  then
            if  $\neg$ CMP_RESPONSE_TIME(ntask,idx+1,ntask) then
                return not schedulable
            end if
        end if
    end for
end if
return schedulable

```

Algorithm A.2 Compute Response Time

```

function CMP_RESPONSE_TIME(ntask, nelements, task)
  while true do
    for all nelements do
      rtime  $\leftarrow$  CEIL(rtimeold/ctaskinter_arrival)  $\times$  ctaskwcet
      ctask  $\leftarrow$  ctask + 1
    end for
    if ntask! = task then
      rtime  $\leftarrow$  CEIL(rtimeold/ntaskinter_arrival)  $\times$  ntaskwcet
    end if
    if rtime  $\geq$  ctaskdeadline then
      return not schedulable
    end if
    if rtimeold  $\geq$  rtime then
      if rtime  $\leq$  taskdeadline then
        return schedulable
      end if
    end if
    rtimeold  $\leftarrow$  rtime
  end while
end function

```

Algorithm A.3 Sufficient test

```

for all nelements do
  if  $ntask_{prio} \geq ctask_{prio}$  then
     $R_{ub} \leftarrow (ntask_{wcet} + sum\_util_{wcet}) / (1 - sum\_util)$ 
    if  $R_{ub} \geq ntask_{deadline}$  then
      return deadline hit
    end if
     $sum\_util \leftarrow ntask_{wcet} / ntask_{inter\_arrival} + sum\_util$ 
     $sum\_util_{wcet} \leftarrow ntask_{wcet} \times (1 - (ntask_{wcet} / ntask_{inter\_arrival}))$ 
  end if
   $R_{ub} \leftarrow (ctask_{wcet} + sum\_util_{wcet}) / (1 - sum\_util)$ 
  if  $R_{ub} \geq ctask_{deadline}$  then
    return deadline hit
  end if
   $sum\_util \leftarrow ctask_{wcet} / ctask_{inter\_arrival}$ 
   $sum\_util_{wcet} \leftarrow ctask_{wcet} \times (1 - ctask_{wcet} / ctask_{inter\_arrival}) + sum\_util_{wcet}$ 
   $ctask \leftarrow ctask + 1$ 
end for
if  $ntask_{prio} \leq (ctask\_prev)_{prio}$  then
   $R_{ub} \leftarrow ntask_{wcet} + sum\_util_{wcet} / (1 - sum\_util)$ 
  if  $R_{ub} \geq ntask_{deadline}$  then
    return deadline hit
  end if
end if
return true

```

B Evaluation Setup

For the evaluation setup, a workstation (i.e. offline system) is connected with one or several (virtualized) embedded systems (i.e. online system) as depicted in figure B.1. The software running on the workstation is responsible for three greater areas during the evaluation phase of the embedded system. First, the generation of test data consists of simulated sensor values as well as the generation of tasks (or tasksets) which can be deployed to the embedded system. Second, the workstation has a control functionality during the actual evaluation phase where single software components on the embedded system can be modified during run-time. Finally, the measured data after an evaluation serves as base for further analysis of the embedded system's performance. Certain variations within the test setup (e.g. software selection) will be given.

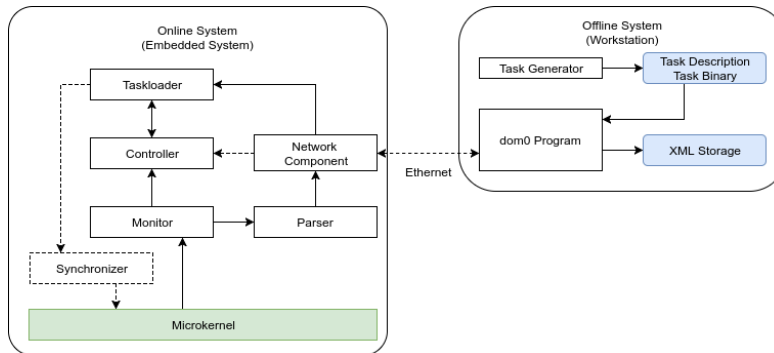


Figure B.1: Detailed toolchain view between offline and online system

B.1 Generating Artificial Task Sets

With the usage of artificially generated task sets, certain corner cases (e.g. overload situations) and their management through the embedded system can be investigated. Especially an analysis of the interplay of all software components as well as the behavior of each component in detail is possible. The overall process and relevant tools for generating artificial task sets for later testing purposes will be described in the following. A task generator is therefore used to generate a wide variety of different task sets (i.e. load scenarios). Each task consists of a description (as mentioned earlier) and the actual execution code (i.e. task binary). The purpose of each test application (i.e. task binary) will be described. Additionally, the process of task distribution (i.e. sending application from taskloader to target platform) will be shown in more detail. In this context the

B Evaluation Setup

target setup (i.e. software components running on the online system) in the terminology of a genode configuration file as well as a short description of possible virtual and physical target platforms will be outlined.

The main purpose of the developed task generator is the generation of tasks (and tasksets) in a target system understandable format (i.e. task description and task binary). Due to the fact that the management mechanism is realized in Genode which itself uses a XML-based configuration for its components, the used task description are also represented by a XML file. The task generator therefore generates a XML representation as task description where a possible alteration of each value of the task model is possible (i.e. however not all combinations make sense). As a result, a fine-grained adjustment of task model attributes is possible manually or fully automated (i.e. randomized). On the other side, the source code of each task (i.e. test application) is available for the task generator. If required, the task generator is able to compile the source code of the needed task to a task binary.

For the inner workings, the task generator consists of several modules with distinct functionality. The most important modules are:

- Task: Representation of a Task Description
- Task Set: A container of several tasks
- Distributor: Socket-based communication
- Monitor: Output of the system responses and additional storing in a database

There exists further modules which are responsible for the configuration of certain operating system components on the target platform like the controller or the type of data generation (e.g. randomized). Additionally parameters (i.e. commandline parameter) for each task binary can be encoded in the task description. This allows not only the modification of timing characteristics of each task but also the modification of its inner working (e.g. varying work load).

The overall processing flow from the generation of tasks over the distribution of tasks to the target system till the receiving of monitored data is shown in algorithm B.1. Starting with opening a new socket-based session via the distributor module of the task generator allows the exchange of tasks and information via a TCP/IP-based communication. The generation process of task descriptions and task binaries can be realized by using a command line interface where predefined test cases can be executed or by using the task generator as a library in its own program. The generation however is using a input file where the relevant tasks are defined (i.e. their descriptions and parameter variations). The controller within in the flexible task management on the target system can be modified (e.g. setting an optimization goal). After the generation is finished and the tasks are distributed to the target system, the actual execution can be separately triggered. During the execution, the target system sends monitoring information back to the offline system (i.e. workstation) where this information can be either printed or stored within a database for later analysis. The session can be explicitly terminated by

the task generator or is implicitly terminated after a complete task execution on the target system.

Algorithm B.1 Basic Session for Controlling the Target Machine (White-Box Test)

```

OPEN_SESSION(ip-address, port)
task-description ← GENERATE_TASKS(task-input.xml)
SEND_TASK-DESCRIPTIONS
SEND_TASK-BINARIES
SET_OPTIMIZER(optimization-goal)
START_EXECUTION
RECEIVE_MONITORED_DATA(storage-type)
CLOSE_SESSION

```

Most of the calls are encapsulated in a high-level operating interface which can be used to fully automate the former described processing flow by a single command line. An example is given in listing B.1 which executes a task set `example.Hey0TaskSet` with one single task. The monitored data is printed on the screen by using the `stdio.StdIOSession`. Due to the specification of an `ip-address`, the distribution of tasks is realized over an socket-based communication channel.

```

./taskgen-cli run -d -t example.Hey0TaskSet -s stdio.StdIOSession ←
172.25.0.1

```

Listing B.1: Example usage of the taskgenerator command line interface

As a next step, it needs to be clarified which concrete content contains a task set. There exists several test applications which are compiled to task binaries and can be transferred to the target system during run-time. The basic idea is to generate as many load situations as possible from a small set of basic tasks. The tasks are mainly for generating load on the target system rather than realizing a distinct functionality. Task descriptions or single parameters can be altered to generate a greater variation. Tasks can be combined in task sets where a task set can consist of a single task. Some applications allow the setting of additional input arguments to control their inner execution (i.e. loop iterations). Again, input arguments can be set within the task description. The table B.1 presents an overview about the available test tasks with a short description about their functionality.

A concrete example of the configuration of such a task (i.e. test application) can be seen in listing B.2. This example demonstrates the assignment of a task to a task set `Problem0`. Several task attributes are set (i.e. key value pairs) where the `period` ← and `priority` attributes indicate a task which will be later scheduled by a fixed priority scheduler.

```

1 class Problem0(TaskSet):
2 def __init__(self):

```

B Evaluation Setup

Task	Description
hey	'Hey' is printed on the command line
namaste	'Namaste' is printed on the command line
tumatmul	All prime numbers until a given parameter will be calculated and the last five numbers are printed on the console.
idle	A while loop .
linpack	Executes a linpack benchmark with an input parameter as problem size.
pi	Number Pi will be calculated iteratively. The argument specifies the number of iterations

Table B.1: Generated Tasks and their descriptions (idp js/mz)

```
3  super().__init__()  
4  
5  self.append( Task( {  
6    # general  
7    "id" : 0, # ignored and set by TaskSet  
8    "criticaltime" : 0,  
9    "executiontime" : 99999999,  
10  
11   # binary  
12   "quota" : "1M",  
13   "pkg" : "hey",  
14   "config" : {},  
15  
16   # frequency  
17   "numberofjobs" : 0,  
18   "period" : 3,  
19  
20   # scheduler  
21   "priority" : 10,  
22  })
```

Listing B.2: Example configuration of a task description with fixed values

Leaving the task generator on the workstation side, the operating system running on the target system needs to be configured accordingly. This is done by Genode's configuration mechanism via so called run-files. A run-file specifies several aspects about the components and their communications as well as controlling the overall build process (i.e. which component is required for a certain use-case). A simple configuration example can be seen in listing B.3. Depending on the components which need to be executed during run-time, a configuration file can contain several entries from type start. In this case however only the scheduling controller will be executed during run-time. For executing the operating system as presented in chapter 6 the configuration is more extensive than the presented example. In common, for each evaluation test presented

within this chapter, a separate configuration file similar to the presented listing will be used.

```

1 set config {
2 <config verbose="yes" prio_levels="128">
3 <parent-provides>
4 <service name="RAM" />
5 <service name="CPU" />
6 </parent-provides>
7 <default-route>
8 <any-service> <parent/> <any-child/> </any-service>
9 </default-route>
10 <start name="sched_controller">
11 <resource name="RAM" quantum="1M" />
12 <provides> <service name="Sched_controller" /> </provides>
13 </start>
14 append config {
15 </config>
16 }

```

Listing B.3: configuration file for operating system running on the target system

This section has covered the generation of artificial task sets as well as common mechanism for configuring the target system. In its heart of the generation process a task generator is used and takes care of the several running states during execution. Furthermore the used test applications/tasks were presented as well as their configuration. The section closes with the configuration mechanism used by genode to setup the target system.

B.2 Autonomous Driving Setup

The highly experimental system approach which is developed in the context of this thesis requires a run-time environment which is able to test the behavior of the system in certain cases. On the other side, these tests can lead to unforeseen consequences which makes it unfeasible to test the system within the context of a real traffic scenario. The presented approach for testing the autonomous driving scenario therefore foresees the usage of a hybrid test setup that was developed in the KIA4SM [91] project. The test setup combines an automotive racing simulation with a physical model car (see section B.2.1). The realized evaluation case targets a possible update of a software component on the target system over the air during run-time by similarly keeping the autonomous driving maneuver robust (see section 7.1.2).

B.2.1 Hybrid Test Bed

The hybrid test bed setup used in this scenario uses a combination of virtual and physical test bed parts. An overview of the overall setup will be given where a short description of the parts as well as the overall communication structure will be explained. After that,

B Evaluation Setup

virtual and physical parts are explained in more detail. The section closes with a listing of exchanged information.

An schematic overview about the overall test bed is given in figure B.2. A workstation serves as a simulation endpoint where simulated sensor data in a virtual driving environment are generated. The communication bus within the setup is Ethernet-based following a standard TCP/IP protocol (i.e. no real-time Ethernet). For the execution of the proposed operating system, there are several hardware/software platforms available (i.e. PandaBoard ES). Additionally, there are further platforms (i.e. Raspberry PI) which equip the low-level hardware elements (i.e. sensors) with an adequate network interface enabling the communication with the other platforms in the network.

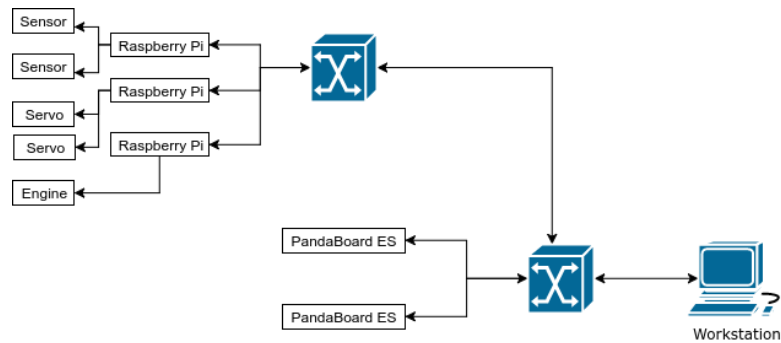


Figure B.2: Schematic View of Evaluation Setup

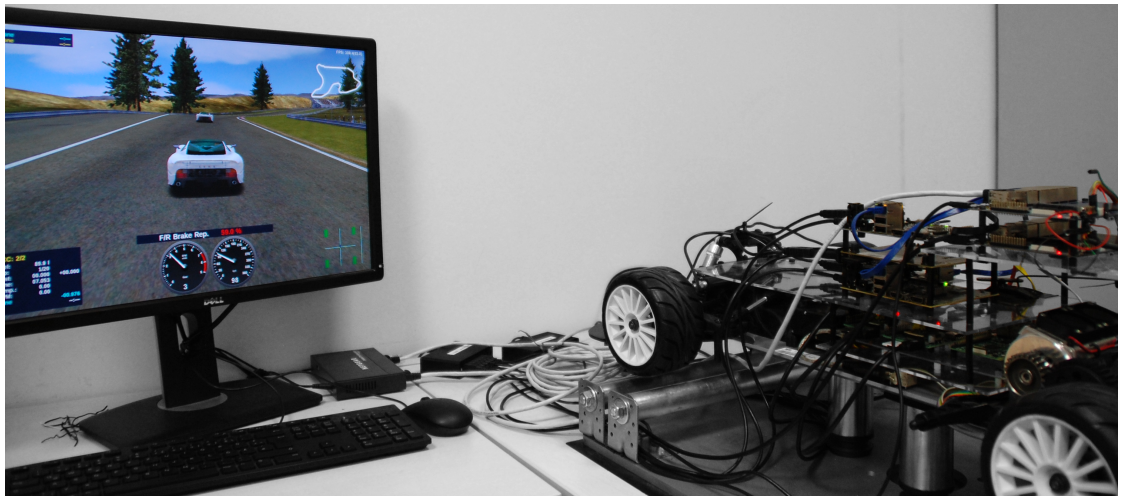


Figure B.3: Test Bed of Evaluation Setup

The used driving environment is based on a racing simulation called Speed Dreams (abbreviated as SD) (i.e. paraten project is TORCS). SD is able to simulate several weather conditions, collision detection (i.e. physics engine) and allows the development of autonomous driving cars (i.e. robots). Especially the last aspect allows the usage of

fully autonomous test cases. An overview about the complete physical setup can be seen in figure B.3.

The introduced hardware platforms serve a model car as electronic control units (see figure B.4). The by Speed Dreams generated virtual sensor values are used to trigger the physical actuating elements (e.g. braking, steering) within the car. Whereby each actuating element is controlled by a corresponding Genode software component. For example, for controlling the several servo motors within the car, a servo controller component was developed. Similar is true for the main motor of the car. Beside the actual controlling of the actuating elements, there exist other software components (i.e. adaptors) which are responsible to translate the simulation values into concrete control events. A given adaptive cruise controller running on a Pandaboard therefore gets its position and distance to a potential front-driving car from the simulation but is also responsible to calculate which force individual physical brakes (e.g. front, left, right) need to hold. With these force values, corresponding commands (e.g. brake_front_left) are executed. Within this setup, information about position, speed, steering and braking are exchanged.

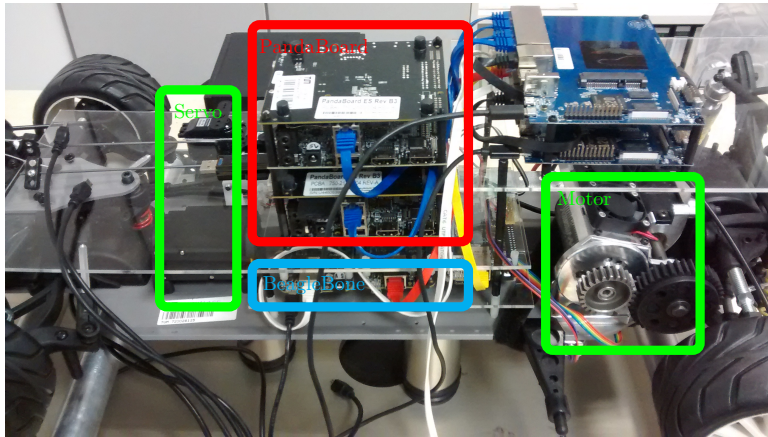


Figure B.4: Physical model car equipped with boards

B.2.2 Installation Scenario Additional Files

```

1  current=int(round(time.time()*1000))
2  session = Dom0_session('10.200.32.121', 3001)
3  name = str(sys.argv[1])
4  session.read_tasks(script_dir + name + '.xml')
5  session.send_descs()
6  session.send_bins()
7  session.start()
8  time.sleep(10)
9  session.live(script_dir + 'xml/' + 'acc_log.xml')
10 i=0
11 while 1:
12     time.sleep(5)

```

B Evaluation Setup

```
13 session.live(script_dir + 'xml/' + str(i) + 'acc_log.xml')
14 i=i+1
```

Listing B.4: Python test for install scenario (workstation)

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!--Hier kommen die Tasks-->
3 <taskset xmlns="http://www.tmsxmlns.com" xmlns:xsi="http://www.w3. ←
   org/2001/XMLSchema-instance" xsi:schemaLocation="http://www. ←
   tmsxmlns.com_taskset.xsd">
4
5   <periodictask>
6     <id>0</id>
7     <executiontime>99999999999</executiontime>
8     <criticaltime>0</criticaltime>
9     <ucfirmrt/>
10    <uawmean>
11      <size>10</size>
12    </uawmean>
13    <priority>127</priority>
14    <numberofjobs>1</numberofjobs>
15    <period>0</period>
16    <offset>0</offset>
17    <quota>5M</quota>
18    <pkg>mbl_adapter</pkg>
19    <config>
20      <network dhcp="yes" />
21      <mosquito ip-address="10.200.32.75" />
22    </config>
23  </periodictask>
24
25 </taskset>
```

Listing B.5: Input file for mbl description

```
1 [2018-03-08 10:21:09] Genode 16.08-87-glabce0d
2 [2018-03-08 10:21:10] [init -> platform_drv] --- Raspberry Pi platform driver ---
3 [2018-03-08 10:21:12] [init -> usb_terminal] PL2303 controller: ready
4 [2018-03-08 10:21:12] [init -> usb_terminal] Manufacturer      : Prolific Technology Inc.
5 [2018-03-08 10:21:12] [init -> usb_terminal] Product             : USB-Serial Controller
6 [2018-03-08 10:21:20] [init -> mbl_client] got IP address 10.200.42.2
7 [2018-03-08 10:21:20] [init -> mbl_client] connected to mosquito server
```

Listing B.6: mbl client capture before install of mbl adapter

```
1 Genode 16.08-88-g7415f76 <local changes>
2 [init -> taskloader] Taskloader_root_component::Taskloader_root_component(Genode:: ←
   Entrypoint*, Genode::Allocator*): Creating root component.
3 [init -> taskloader] Main::Main(Genode::Entrypoint&): task-manager: Hello!
4 [init -> taskloader] virtual Taskloader_session_component* Taskloader_root_component:: ←
   _create_session(const char*): Creating Taskloader session.
5 [init -> dom0-HW] int main(int, char**): dom0: Hello!
6 [init -> dom0-HW] Dom0_server::Dom0_server::Dom0_server(): DHCP network ...
7 [init -> usb_drv] netif.info: register 'sm5c95xx' at usb-ehci-omap-1.1, sm5c95xx USB 2.0 ←
   Ethernet, 0a:02:00:00:00:03
```

B.2 Autonomous Driving Setup

```
8 [init -> nic_bridge] --- NIC bridge started (mac=0a:02:00:00:00:03) ---
9 [init -> dom0-HW] Dom0_server::Dom0_server::Dom0_server(): Waiting 10s for ip ↵
  assignement
10 [2018-03-08 10:21:21] [init -> dom0-HW] got IP address 10.200.32.121
11 [2018-03-08 10:21:28] [init -> dom0-HW] Listening...
12 [2018-03-08 10:21:28] [init -> dom0-HW] Got connection from 124.254.79.32
```

Listing B.7: dom0 hw install of mbl adapter

```
1 [2018-03-08 10:24:59] [init -> dom0-HW] void Dom0_server::Dom0_server:: ↵
  Child_starter_thread::do_send_descs(int): Ready to receive task description.
2 [2018-03-08 10:24:59] [init -> dom0-HW] Ready to receive XML of size 617.
3 [2018-03-08 10:24:59] [init -> dom0-HW] void Dom0_server::Dom0_server:: ↵
  Child_starter_thread::do_send_descs(int): Received XML. Initializing tasks.
4 [2018-03-08 10:24:59] [init -> sched_controller] Update Rq_buffer for core 1!
5 [2018-03-08 10:24:59] [init -> taskloader] id: 0, name: 00.mbl_adapter, prio: 127, ↵
  deadline: 0, wctet: 3567587327, period: 0
6 [2018-03-08 10:25:00] [init -> sched_controller] Task with name 00.mbl_adapter, is now ↵
  enqueued to run queue 1
7 [2018-03-08 10:25:00] [init -> sched_controller] Rq is empty, Task set is schedulable!
8 [2018-03-08 10:25:00] [init -> sched_controller] Sched_controller (enq): Task 00. ↵
  mbl_adapter was rta analyzed
9 [2018-03-08 10:25:00] [init -> sched_controller] New element inserted to buffer at ↵
  position 0 with pointer e010
10 [2018-03-08 10:25:00] [init -> dom0-HW] void Dom0_server::Dom0_server::serve(): Done ↵
  SEND_DESCS. Took: 487
11 [2018-03-08 10:25:00] [init -> dom0-HW] void Dom0_server::Dom0_server:: ↵
  Child_starter_thread::do_send_binaries(int): Ready to receive binaries.
12 [2018-03-08 10:25:00] [init -> dom0-HW] 1 binary to be sent.
13 [2018-03-08 10:25:00] [init -> dom0-HW] Got binary 'mbl_adapter' of size 51224.
14 [2018-03-08 10:25:00] [init -> dom0-HW] void Dom0_server::Dom0_server::serve(): Done ↵
  SEND_BINARIES. Took: 178
15 [2018-03-08 10:25:00] [init -> dom0-HW] void Dom0_server::Dom0_server:: ↵
  Child_starter_thread::do_start(int): Starting tasks.
16 [2018-03-08 10:25:00] [init -> dom0-HW] void Dom0_server::Dom0_server::serve(): Done ↵
  START. Took: 130
17 [2018-03-08 10:25:00] [init -> taskloader] virtual void Task::Child_start_thread::entry ↵
  (): Starting task 00.mbl_adapter
18 [2018-03-08 10:25:00] [init -> taskloader] bool Task::jobs_done(): iteration: 0 num jobs ↵
  : 1 name: 00.mbl_adapter
19 [2018-03-08 10:25:05] [init -> taskloader -> 00.mbl_adapter] got IP address ↵
  10.200.32.122
20 [2018-03-08 10:25:05] [init -> taskloader -> 00.mbl_adapter] connected to mosquito ↵
  server
21 [2018-03-08 10:25:05] [init -> taskloader -> 00.mbl_adapter] unknown topic: savm/car/0/ ↵
  steer
```

Listing B.8: dom0 hw capture after install of mbl adapter

```
1 [2018-03-08 10:25:07] [init -> mbl_client] setMotorSpeedAbs finished with 0
2 [2018-03-08 10:25:07] [init -> mbl_client] powerpct 23
3 [2018-03-08 10:25:07] [init -> mbl_client] setCommand 230
4 [2018-03-08 10:25:07] [init -> mbl_client] setMotorSpeedAbs finished with 0
```

Listing B.9: mbl client capture after install of mbl adapter

```
1 current=int(round(time.time()*1000))
2 session = Dom0_session('10.200.44.16', 3001)
3 name = str(sys.argv[1])
4 counter=0
5 while 1:
6     session.read_tasks(script_dir + name + str(counter) + '.xml')
7     session.send_descs()
8     session.send_bins()
9     session.start()
```

B Evaluation Setup

```
10 time.sleep(30)
11 session.live(script_dir + 'xml/' + 'update' + str(counter) + '_log ←
    .xml')
12 counter=counter+1
13 session.stop()
```

Listing B.10: python test for update scenario (workstation)

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!--Hier kommen die Tasks-->
3 <taskset xmlns="http://www.tmsxmlns.com" xmlns:xsi="http://www.w3. ←
    org/2001/XMLSchema-instance" xsi:schemaLocation="http://www. ←
    tmsxmlns.com-taskset.xsd">
4
5     <periodictask>
6         <id>0</id>
7         <executiontime>99999999999</executiontime>
8         <criticaltime>0</criticaltime>
9         <ucfirmrt />
10        <uawmean>
11            <size>10</size>
12        </uawmean>
13        <priority>127</priority>
14        <numberofjobs>1</numberofjobs>
15        <period>0</period>
16        <offset>0</offset>
17        <quota>1M</quota>
18        <pkg>pi</pkg>
19        <config>
20            <arg1>4</arg1>
21        </config>
22    </periodictask>
23
24 </taskset>
```

Listing B.11: Input task description for one pi update case

Own Publications

- [50] Sebastian Eckl, Daniel Krefft, and Uwe Baumgarten. “COFAT 2015 - KIA4SM - Cooperative Integration Architecture for Future Smart Mobility Solutions”. In: *Conference on Future Automotive Technology*. 2015.
- [51] Sebastian Eckl, Daniel Krefft, and Uwe Baumgarten. “Migration of Components and Processes as Means for Dynamic Reconfiguration in Distributed Embedded Real-Time Operating Systems”. In: *OSPERT 2017*. 2017.
- [90] Daniel Krefft and Uwe Baumgarten. “Flexible Scheduling of Tasks for Self-Adaptation in Mixed-Critical Automotive Systems”. In: *Organic Computing Doctoral Dissertation Colloquium 2015*. Ed. by Sven Tomforde and Bernhard Sick. Intelligent embedded systems 7. Augsburg, Germany: kassel university press GmbH, 2015, pp. 147–153.
- [91] Daniel Krefft, Sebastian Eckl, and Uwe Baumgarten. *KIA4SM - Kooperative Integrationsarchitektur Für Zukünftige Smart Mobility Lösungen*. Gesellschaft für Informatik - Fachgruppe für Betriebssysteme, 2017.

Advised Theses

- [33] Gurusiddesha Chandrasekhara. “Design and Prototypical Implementation of a High-Level Synchronization Component for Dynamic Updates of Task Run Queues in L4 Fiasco.OC/Genode”. MA thesis. Technische Universität München, 2016.
- [52] Stefan Edinger. *Erweiterung von L4 Fiasco.OC/Genode Zur Verwaltung von Koexistenten Scheduling Strategien Auf Einem Eingebetteten Echtzeitfähigen Mehrkernsystem*. Interdisziplinäres Projekt im Anwendungsfach Elektro- und Informationstechnik. Technische Universität München, 2016.
- [53] Stefan Edinger. “Improved Task Management for Multi-Core Real-Time Systems in an Automotive Environment”. MA thesis. Technische Universität München, Aug. 15, 2017.
- [56] Boris Espinoza-Kalchev. “Comparison of the PikeOS Hypervisor and the L4 Fiasco.OC/Genode: Development of a PikeOS Emulation Layer and Porting on Xilinx Zynq-7000 SoC”. MA thesis. Technische Universität München, Apr. 15, 2016.
- [64] Georg Guba. “Port and Extension of a Toolchain Regarding Machine Learning Supported Schedulability Analysis in Distributed Embedded Real-Time Systems”. MA thesis. Technische Universität München, Apr. 15, 2016.
- [66] Robert Häcker. “Design of an OC-Based Method for Efficient Synchronization of L4 Fiasco.OC Microkernel Tasks”. BA thesis. Technische Universität München, June 15, 2015.
- [68] Valentin Hauner. “Extension of the Fiasco.OC Microkernel with Context-Sensitive Scheduling Abilities for Safety-Critical Applications in Embedded Systems”. BA thesis. Technische Universität München, Oct. 15, 2014.
- [70] Mathias Helminger. “Dynamic Realtime Scheduling of Quasi-Periodic Tasks for Embedded Systems Testing on Linux”. MA thesis. Technische Universität München, 2014.
- [108] Pritpal Singh Multani. “Analysis and Evaluation of Scheduling Strategies for Safety Critical Automotive Systems”. MA thesis. Technische Universität München, Dec. 15, 2013.
- [112] Barbara Niedermeier. “Knowledge-Based Algorithms for Dynamic Task Management in Embedded Multi-Core Systems”. MA thesis. Technische Universität München, 2017.

Advised Theses

- [113] Paul Nieleck. “Design and Prototypical Implementation of an OC-Based Controller-Stack for Optimizing Mixed-Critical Thread Scheduling in L4 Fiasco.OC/Genode”. MA thesis. Technische Universität München, Oct. 17, 2016.
- [119] Alexander Reisner. “Extension of the Genode OS Framework by a Component for Runtime-Monitoring of a Real-Time Operating System”. BA thesis. Technische Universität München, Dec. 15, 2015.
- [129] Jonas Sticha. “Validating the Real-Time Capabilities of the ROS Communication Middleware”. BA thesis. Technische Universität München, June 15, 2014.

Bibliography

- [1] Luca Abeni and Giorgio Buttazzo. “Integrating Multimedia Applications in Hard Real-Time Systems”. In: *Real-Time Systems Symposium, 1998. Proceedings. The 19th IEEE*. 00771. IEEE, 1998, pp. 4–13.
- [2] Luca Abeni and Giorgio Buttazzo. “Resource reservation in dynamic real-time systems”. In: *Real-Time Systems* 27.2 (2004), pp. 123–167.
- [3] Luis Almeida et al. “A dynamic scheduling approach to designing flexible safety-critical systems”. In: Proceedings of the 7th ACM & IEEE international conference on Embedded software. 2007, pp. 67–74.
- [4] James H. Anderson, Sanjoy K. Baruah, and Björn B. Brandenburg. “Multicore operating-system support for mixed criticality”. In: *Proceedings of the Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*. 00041. Citeseer, 2009.
- [5] Mikael Åsberg and Thomas Nolte. “Towards a user-mode approach to partitioned scheduling in the seL4 microkernel”. In: *ACM SIGBED Review* 10.3 (2013), pp. 15–22.
- [6] Mikael Åsberg, Thomas Nolte, and Shinpei Kato. “Towards partitioned hierarchical real-time scheduling on multi-core processors”. In: *ACM SIGBED Review* 11.2 (2014), pp. 13–18.
- [7] Mikael Åsberg et al. “Towards hierarchical scheduling in AUTOSAR”. In: Emerging Technologies & Factory Automation, 2009. ETFA 2009. IEEE Conference on. IEEE, 2009, pp. 1–8.
- [8] Neil Audsley et al. “Real-time system scheduling”. In: *Predictably Dependable Computing Systems*. Springer, 1995, pp. 41–52.
- [9] AUTOSAR. *Virtual Functional Bus*. Dec. 8, 2017.
- [10] Algirdas Avizienis et al. “Basic concepts and taxonomy of dependable and secure computing”. In: *IEEE transactions on dependable and secure computing* 1.1 (2004), pp. 11–33.
- [11] S. Baruah and S. Vestal. “Schedulability Analysis of Sporadic Tasks with Multiple Criticality Specifications”. In: Euromicro Conference on Real-Time Systems, 2008. ECRTS 08. July 2008, pp. 147–155.
- [12] Sanjoy K. Baruah, Alan Burns, and Robert I. Davis. “Response-time analysis for mixed criticality systems”. In: Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd. IEEE, 2011, pp. 34–43.

Bibliography

- [13] Sanjoy Baruah et al. “Scheduling Real-Time Mixed-Criticality Jobs”. In: *IEEE Transactions on Computers* 61.8 (Aug. 2012), pp. 1140–1152.
- [14] Sanjoy Baruah et al. “Mixed-criticality scheduling on multiprocessors”. In: *Real-Time Systems* 50.1 (Jan. 2014). 00058, pp. 142–177.
- [15] Andrew Baumann et al. “The multikernel: a new OS architecture for scalable multicore systems”. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. ACM, 2009, pp. 29–44.
- [16] Jean-Luc Bechennec et al. “Trampoline an open source implementation of the osek/vdx rtos specification”. In: *Emerging Technologies and Factory Automation, 2006. ETFA '06. IEEE Conference on*. IEEE, 2006, pp. 62–69.
- [17] Klaus Becker, Marc Zeller, and Gereon Weiss. “Towards Efficient On-line Schedulability Tests for Adaptive Networked Embedded Real-time Systems.” In: *PECCS*. 2012, pp. 440–449.
- [18] Marko Bertogna. “Evaluation of existing schedulability tests for global EDF”. In: Parallel Processing Workshops, 2009. ICPPW09. International Conference on. IEEE, 2009, pp. 11–18.
- [19] Enrico Bini et al. “A Response-Time Bound in Fixed-Priority Scheduling with Arbitrary Deadlines”. In: *IEEE Transactions on Computers* 58.2 (2009), pp. 279–286.
- [20] Bernard Blackham et al. “Timing analysis of a protected operating system kernel”. In: *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*. IEEE, 2011, pp. 339–348.
- [21] Thomas Bloor. *The Automotive Shift to Software-Defined, Consolidated Controller Architectures*. <http://qnxauto.blogspot.com/2016/10/automotive-shifting-software-defined.html>. Accessed: 2018-07-21.
- [22] Scott A. Brandt et al. “Dynamic integrated scheduling of hard real-time, soft real-time, and non-real-time processes”. In: *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*. IEEE, 2003, pp. 396–407.
- [23] Jürgen Branke et al. “Organic Computing Addressing Complexity by Controlled Self-Organization”. In: IEEE, Nov. 2006, pp. 185–191.
- [24] Martin Buechel et al. “An Automated Electric Vehicle Prototype Showing New Trends in Automotive Architectures”. In: IEEE, Sept. 2015, pp. 1274–1279.
- [25] A. Burns and A.J. Wellings. *Real-Time Systems and Programming Languages: Ada 95, Real-Time Java, and Real-Time POSIX*. International computer science series. 01579 LCCB: 2001016408. Addison-Wesley, 2001.
- [26] Alan Burns and Rob Davis. “Mixed criticality systems-a review”. In: *Department of Computer Science, University of York, Tech. Rep* (2013).
- [27] Alan Burns, Andy J. Wellings, and Fengxiang Zhang. “Combining EDF and FP Scheduling: Analysis and Implementation in Ada 2005.” In: *Ada-Europe*. Springer, 2009, pp. 119–133.

- [28] Alan Burns et al. “The meaning and role of value in scheduling flexible real-time systems”. In: *Journal of systems architecture* 46.4 (2000), pp. 305–325.
- [29] G. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Real-Time Systems Series. 01953 LCCB: 2011937234. Springer US, 2011.
- [30] Giorgio C. Buttazzo. “Rate monotonic vs. EDF: Judgment day”. In: *Embedded Software*. Springer, 2003, pp. 67–83.
- [31] Giorgio Buttazzo and Paolo Gai. “Efficient EDF implementation for small embedded systems”. In: *Proc. of the 2nd Int. Workshop on Operating Systems Platforms for Embedded Real-Time applications*, Dresden, Germany. 2006.
- [32] Emre Cakar et al. “Towards a quantitative notion of self-organisation”. In: *Evolutionary Computation*, 2007. CEC 2007. IEEE Congress on. IEEE, 2007, pp. 4222–4229.
- [33] Gurusiddhesha Chandrasekhara. “Design and Prototypical Implementation of a High-Level Synchronization Component for Dynamic Updates of Task Run Queues in L4 Fiasco.OC/Genode”. MA thesis. Technische Universität München, 2016.
- [34] Ya-Shu Chen, Han Chiang Liao, and Ting-Hao Tsai. “Online Real-Time Task Scheduling in Heterogeneous Multicore System-on-a-Chip”. In: *IEEE Transactions on Parallel and Distributed Systems* 24.1 (Jan. 2013), pp. 118–130.
- [35] Betty H. C. Cheng, ed. *Software engineering for self-adaptive systems*. Lecture notes in computer science 5525. Berlin ; New York: Springer, 2009. 260 pp.
- [36] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. “Making self-adaptation an engineering reality”. In: *Self-star properties in complex information systems*. Springer, 2005, pp. 158–173.
- [37] Shang-Wen Cheng et al. “Using architectural style as a basis for system self-repair”. In: *Software Architecture*. Springer, 2002, pp. 45–59.
- [38] Autonomic Computing et al. “An Architectural Blueprint for Autonomic Computing”. In: *IBM White Paper* 31 (2006).
- [39] AUTOSAR development cooperation. *Adaptive Platform*. URL: <https://www.autosar.org/standards/adaptive-platform/> (visited on 11/07/2017).
- [40] AUTOSAR development cooperation. *AUTOSAR*. URL: <https://195.234.139.136/> (visited on 07/28/2018).
- [41] *CPS Cyber Physical Systems*. URL: <http://addi-data.com/cps-cyber-physical-systems/> (visited on 05/29/2018).
- [42] *Cyber-Physical Systems Public Working Group*. URL: <https://pages.nist.gov/cpspwg/> (visited on 11/22/2017).
- [43] Robert I. Davis and Alan Burns. “Hierarchical fixed priority pre-emptive scheduling”. In: *Real-Time Systems Symposium*, 2005. RTSS 2005. 26th IEEE International. IEEE, 2005, 10–pp.

Bibliography

- [44] Robert I. Davis and Alan Burns. “A survey of hard real-time scheduling for multiprocessor systems”. In: *ACM Comput. Surv.* 43.4 (Oct. 2011), 35:1–35:44.
- [45] Rogério De Lemos et al. “Software Engineering for Self-adaptive Systems: Research Challenges in the Provision of Assurances”. In: *Software Engineering for Self-Adaptive Systems II*. Springer, 2013, pp. 1–32.
- [46] Tom De Wolf and Tom Holvoet. “Towards a Methodology for Engineering Self-Organising Emergent Systems”. In: *Frontiers in Artificial Intelligence and Applications* 135 (2005), p. 18.
- [47] Sanjay R. Deshpande. “Chapter 5 - Design Considerations for Multicore SoC Interconnections”. In: *Real World Multicore Embedded Systems*. Ed. by Bryon Moyer. Oxford: Newnes, 2013, pp. 117–197.
- [48] C. Ebert and J. Favaro. “Automotive Software”. In: *IEEE Software* 34.3 (May 2017), pp. 33–39.
- [49] Christof Ebert. “Functional Safety with ISO 26262”. Oct. 18, 2016.
- [50] Sebastian Eckl, Daniel Krefft, and Uwe Baumgarten. “COFAT 2015 - KIA4SM - Cooperative Integration Architecture for Future Smart Mobility Solutions”. In: *Conference on Future Automotive Technology*. 2015.
- [51] Sebastian Eckl, Daniel Krefft, and Uwe Baumgarten. “Migration of Components and Processes as Means for Dynamic Reconfiguration in Distributed Embedded Real-Time Operating Systems”. In: *OSPERT 2017*. 2017.
- [52] Stefan Edinger. *Erweiterung von L4 Fiasco.OC/Genode Zur Verwaltung von Koexistenten Scheduling Strategien Auf Einem Eingebetteten Echtzeitfähigen Mehrkernsystem*. Interdisziplinäres Projekt im Anwendungsfach Elektro- und Informationstechnik. Technische Universität München, 2016.
- [53] Stefan Edinger. “Improved Task Management for Multi-Core Real-Time Systems in an Automotive Environment”. MA thesis. Technische Universität München, Aug. 15, 2017.
- [54] Kevin Elphinstone and Gernot Heiser. “From L3 to seL4 what have we learnt in 20 years of L4 microkernels?” In: ACM Press, 2013, pp. 133–150.
- [55] *ERIKA Enterprise — Open Source RTOS OSEK/VDX Kernel*. URL: <http://erika.tuxfamily.org/drupal/> (visited on 11/07/2017).
- [56] Boris Espinoza-Kalchev. “Comparison of the PikeOS Hypervisor and the L4 Fiasco.OC/Genode: Development of a PikeOS Emulation Layer and Porting on Xilinx Zynq-7000 SoC”. MA thesis. Technische Universität München, Apr. 15, 2016.
- [57] Peter Fischer et al. “Ensuring correct self-reconfiguration in safety-critical applications by verified result checking”. In: Proceedings of the 2011 workshop on Organic computing. ACM, 2011, pp. 3–12.

- [58] Tom Fleming and Alan Burns. “Incorporating the notion of importance into mixed criticality systems”. In: *Proc. 2nd Workshop on Mixed Criticality Systems (WMC), RTSS*. 2014, pp. 33–38.
- [59] Katharina Gilles et al. “Proteus hypervisor: Full virtualization and paravirtualization for multi-core embedded systems”. In: *International Embedded Systems Symposium*. Springer, 2013, pp. 293–305.
- [60] John Greenough. *The massive internet-connected car market explained in one infographic*. URL: <https://www.businessinsider.de/complete-connected-infographic-2015-2> (visited on 04/27/2018).
- [61] Stefan Groesbrink and Luis Almeida. “A criticality-aware mapping of real-time virtual machines to multi-core processors”. In: *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*. IEEE, 2014, pp. 1–9.
- [62] Stefan Groesbrink, Simon Oberthür, and Daniel Baldin. “Architecture for adaptive resource assignment to virtualized mixed-criticality real-time systems”. In: *ACM SIGBED Review* 10.1 (2013), pp. 18–23.
- [63] Stefan Groesbrink et al. “Towards Certifiable Adaptive Reservations for Hypervisor-Based Virtualization”. In: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*. IEEE, 2014, pp. 13–24.
- [64] Georg Guba. “Port and Extension of a Toolchain Regarding Machine Learning Supported Schedulability Analysis in Distributed Embedded Real-Time Systems”. MA thesis. Technische Universität München, Apr. 15, 2016.
- [65] Zhishan Guo, Luca Santinelli, and Kecheng Yang. “Edf schedulability analysis on mixed-criticality systems with permitted failure probability”. In: *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2015 IEEE 21st International Conference on*. IEEE, 2015, pp. 187–196.
- [66] Robert Häcker. “Design of an OC-Based Method for Efficient Synchronization of L4 Fiasco.OC Microkernel Tasks”. BA thesis. Technische Universität München, June 15, 2015.
- [67] Hermann Härtig and Michael Roitzsch. “Ten years of research on L4-based real-time systems”. In: *Proceedings of the 8th Real-Time Linux Workshop*. 2006.
- [68] Valentin Hauner. “Extension of the Fiasco.OC Microkernel with Context-Sensitive Scheduling Abilities for Safety-Critical Applications in Embedded Systems”. BA thesis. Technische Universität München, Oct. 15, 2014.
- [69] Gernot Heiser. “The role of virtualization in embedded systems”. In: *ACM Press*, 2008, pp. 11–16.
- [70] Mathias Helminger. “Dynamic Realtime Scheduling of Quasi-Periodic Tasks for Embedded Systems Testing on Linux”. MA thesis. Technische Universität München, 2014.

Bibliography

- [71] Jonathan L. Herman et al. “RTOS support for multicore mixed-criticality systems”. In: *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*. IEEE, 2012, pp. 197–208.
- [72] Jens Hildebrandt, Frank Golasowski, and Dirk Timmermann. “Scheduling co-processor for enhanced least-laxity-first scheduling in hard real-time systems”. In: *Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on*. IEEE, 1999, pp. 208–215.
- [73] Huang-Ming Huang, Christopher Gill, and Chenyang Lu. “Implementation and Evaluation of Mixed-Criticality Scheduling Approaches for Periodic Tasks”. In: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*. IEEE, 2012, pp. 23–32.
- [74] Huang-Ming Huang, Christopher Gill, and Chenyang Lu. “Implementation and Evaluation of Mixed-Criticality Scheduling Approaches for Sporadic Tasks”. In: *ACM Transactions on Embedded Computing Systems* 13 (4s Apr. 1, 2014), pp. 1–25.
- [75] ISO. *ISO 26262-1-Road Vehicles Functional Safety Part 1 Vocabulary*. Technical report, International Organization for Standardization/Technical Committee 22 (ISO/TC 22), 2011.
- [76] Mathieu Jan, Lilia Zaourar, and Maurice Pitel. “Maximizing the execution rate of low-criticality tasks in mixed criticality system”. In: *Proc. WMC, RTSS (2013)*, pp. 43–48.
- [77] Roland Kammerer. *Linux in safety-critical applications*. OSADL Heidelberg, 2011.
- [78] Owen R. Kelly, Hakan Aydin, and Baoxian Zhao. “On Partitioned Scheduling of Fixed-Priority Mixed-Criticality Task Sets”. In: IEEE, Nov. 2011, pp. 1051–1059.
- [79] J. O. Kephart and D. M. Chess. “The vision of autonomic computing”. In: *Computer* 36.1 (2003), pp. 41–50.
- [80] Nima Moghaddami Khalilzad, Moris Behnam, and Thomas Nolte. “Implementation of the Multi-Level Adaptive Hierarchical Scheduling Framework”. In: *9th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERS)*. 2013, pp. 11–19.
- [81] Nima Moghaddami Khalilzad, Thomas Nolte, and Moris Behnam. “Towards adaptive hierarchical scheduling of overloaded real-time systems”. In: *Industrial Embedded Systems (SIES), 2011 6th IEEE International Symposium on*. IEEE, 2011, pp. 39–42.
- [82] Olaf Kindel and Mario Friedrich. *Softwareentwicklung Mit AUTOSAR: Grundlagen*. Vol. 1. dpunkt, 2009.
- [83] Gerwin Klein et al. “seL4: Formal verification of an OS kernel”. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 207–220.

- [84] Florian Kluge. “Autonomic- Und Organic-Computing-Techniken Für Eingebettete Echtzeitsysteme”. Doctoralthesis. Universität Augsburg, 2011.
- [85] Florian Kluge, Markus Neuerburg, and Theo Ungerer. “Utility-based scheduling of (m, k)-firm real-time task sets”. In: *Architecture of Computing Systems–ARCS 2015*. Springer, 2015, pp. 201–211.
- [86] Florian Kluge et al. “A Two-Layered Management Architecture for Building Adaptive Real-Time Systems”. In: *Software Technologies for Embedded and Ubiquitous Systems*. Ed. by Uwe Brinkschulte, Tony Givargis, and Stefano Russo. 5287 vols. Springer Berlin Heidelberg, Jan. 2008, pp. 126–137.
- [87] Florian Kluge et al. “An Operating System Architecture for Organic Computing in Embedded Real-Time Systems”. In: *Autonomic and Trusted Computing*. Ed. by Chunming Rong et al. 5060 vols. Springer Berlin Heidelberg, Jan. 2008, pp. 343–357.
- [88] Hermann Kopetz. *Real-Time Systems - Design Principles for Distributed Embedded Applications*. Real-Time Systems Series. Springer, 2011.
- [89] Jeff Kramer and Jeff Magee. “Self-managed systems: an architectural challenge”. In: *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 259–268.
- [90] Daniel Krefft and Uwe Baumgarten. “Flexible Scheduling of Tasks for Self-Adaptation in Mixed-Critical Automotive Systems”. In: *Organic Computing Doctoral Dissertation Colloquium 2015*. Ed. by Sven Tomforde and Bernhard Sick. Intelligent embedded systems 7. Augsburg, Germany: kassel university press GmbH, 2015, pp. 147–153.
- [91] Daniel Krefft, Sebastian Eckl, and Uwe Baumgarten. *KIA4SM - Kooperative Integrationsarchitektur Für Zukünftige Smart Mobility Lösungen*. Gesellschaft für Informatik - Fachgruppe für Betriebssysteme, 2017.
- [92] Avinash Kundaliya. “Design and Prototypical Implementation of a Toolchain for Offline Task-to-Machine Mapping in Distributed Embedded Systems”. MA thesis. Technische Universität München, Oct. 15, 2015.
- [93] Adam Lackorzynski et al. “Flattening hierarchical scheduling”. In: Proceedings of the tenth ACM international conference on Embedded software. ACM, 2012, pp. 93–102.
- [94] Barbara Lange. “Vorhersagbar”. In: *Magazin für professionelle Informationstechnik iX* 10 (2013), p. 4.
- [95] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. Second edition. Cambridge, Massachusetts: MIT Press, 2017. 537 pp.
- [96] Ye Li, Richard West, and Eric Missimer. “The Quest-V Separation Kernel for Mixed Criticality Systems”. In: *arXiv:1310.6298 [cs]* (Oct. 23, 2013). arXiv: 1310.6298.

Bibliography

- [97] Jochen Liedtke. “Improving IPC by kernel design”. In: *ACM SIGOPS Operating Systems Review* 27.5 (1993), pp. 175–188.
- [98] Jochen Liedtke. *On micro-kernel construction*. Vol. 29. 5. ACM, 1995.
- [99] Anna Lyons and Gernot Heiser. “Mixed-Criticality Support in a High-Assurance, General-Purpose Microkernel”. In: WMC. 2014, pp. 9–9.
- [100] Alejandro Masrur, Samarjit Chakraborty, and G. Farber. “Constant-time admission control for partitioned EDF”. In: *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*. IEEE, 2010, pp. 34–43.
- [101] Nicholas Mc Guire. “Linux for Safety Critical Systems in IEC 61508 Context”. In: *Proceedings of the Ninth Real-Time Linux Workshop in Linz*. 2007.
- [102] Detlev Mohr, Hans-Werner Kaas, and Paul Gao. *Automotive Revolution: Perspective towards 2030: How the Convergence of Disruptive Technology-Driven Trends Could Transform the Auto Industry*. McKinsey and Company, 2016.
- [103] Malcolm S. Mollison et al. “Mixed-criticality real-time scheduling for multicore systems”. In: *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*. IEEE, 2010, pp. 1864–1871.
- [104] Aurélien Monot et al. “Multicore scheduling in automotive ECUs”. In: *Embedded Real Time Software and Systems-ERTSS 2010*. 2010.
- [105] Bryon Moyer. “Chapter 1 - Introduction and Roadmap”. In: *Real World Multicore Embedded Systems*. Ed. by Bryon Moyer. Oxford: Newnes, 2013, pp. 1–10.
- [106] Bryon Moyer. “Chapter 6 - Operating Systems in Multicore Platforms”. In: *Real World Multicore Embedded Systems*. Ed. by Bryon Moyer. Oxford: Newnes, 2013, pp. 199–226.
- [107] Christian Müller-Schloer, Hartmut Schmeck, and Theo Ungerer, eds. *Organic Computing — A Paradigm Shift for Complex Systems*. Basel: Springer Basel, 2011.
- [108] Pritpal Singh Multani. “Analysis and Evaluation of Scheduling Strategies for Safety Critical Automotive Systems”. MA thesis. Technische Universität München, Dec. 15, 2013.
- [109] *Multicore and Virtualization in Automotive Environments*. URL: <https://www.edn.com/design/automotive/4399434/Multicore-and-virtualization-in-automotive-environments> (visited on 05/20/2018).
- [110] Nicolas Navet et al. “Multi-source and multicore automotive ECUs-OS protection mechanisms and scheduling”. In: *Industrial Electronics (ISIE), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 3734–3741.
- [111] Mircea Negrean, Simon Schliecker, and Rolf Ernst. “Response-time analysis of arbitrarily activated tasks in multiprocessor systems with shared resources”. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2009, pp. 524–529.

- [112] Barbara Niedermeier. “Knowledge-Based Algorithms for Dynamic Task Management in Embedded Multi-Core Systems”. MA thesis. Technische Universität München, 2017.
- [113] Paul Nieleck. “Design and Prototypical Implementation of an OC-Based Controller-Stack for Optimizing Mixed-Critical Thread Scheduling in L4 Fiasco.OC/Genode”. MA thesis. Technische Universität München, Oct. 17, 2016.
- [114] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. “Architecture-based runtime software evolution”. In: *Proceedings of the 20th international conference on Software engineering*. IEEE Computer Society, 1998, pp. 177–186.
- [115] Peyman Oreizy et al. “An architecture-based approach to self-adaptive software”. In: *IEEE Intelligent Systems and Their Applications* 14.3 (1999), pp. 54–62.
- [116] Hewlett Peckard. *Connected Cars*. https://infographic.statista.com/normal/infografik_4717_connected_cars_n.jpg. [Online; accessed 26-April-2018].
- [117] R. H Pierce, Great Britain, and Health and Safety Executive. *Preliminary assessment of Linux for safety related systems*. OCLC: 319976931. Sudbury: HSE Books, 2002.
- [118] Binoy Ravindran, E. Douglas Jensen, and Peng Li. “On recent advances in time/utility function real-time scheduling and resource management”. In: *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*. IEEE, 2005, pp. 55–60.
- [119] Alexander Reisner. “Extension of the Genode OS Framework by a Component for Runtime-Monitoring of a Real-Time Operating System”. BA thesis. Technische Universität München, Dec. 15, 2015.
- [120] Urban Richter et al. “Towards a generic observer/controller architecture for organic computing”. In: *Informatik (2006)*, pp. 112–119.
- [121] Matthias Rohr et al. *A classification scheme for self-adaptation research*. 2006.
- [122] Vahid Salmani, Saman Taghavi Zargar, and Mahmoud Naghibzadeh. “A modified maximum urgency first scheduling algorithm for real-time tasks”. In: *Transactions on Engineering, Computing and Technology, ISSN 1305 5313* (2005), pp. 19–23.
- [123] Rodrigo Santos, Giuseppe Lipari, and Enrico Bini. “Efficient on-line schedulability test for feedback scheduling of soft real-time tasks under fixed-priority”. In: *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS08*. IEEE, 2008, pp. 227–236.
- [124] Francois Santy et al. “Relaxing mixed-criticality scheduling strictness for task sets scheduled with fp”. In: *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*. IEEE, 2012, pp. 155–165.
- [125] Frank Schirrmeister. “Chapter 3 - Multicore Architectures”. In: *Real World Multicore Embedded Systems*. Ed. by Bryon Moyer. Oxford: Newnes, 2013, pp. 33–73.

Bibliography

- [126] Philipp Schlei. *Integration of Highly-Automated Driving Functions with Fail-Operational Properties*. 2017.
- [127] Hartmut Schmeck et al. “Adaptivity and self-organization in organic computing systems”. In: *ACM Transactions on Autonomous and Adaptive Systems* 5.3 (Sept. 2010), pp. 1–32.
- [128] David B Stewart and Pradeep K Khosla. “Real-time scheduling of dynamically reconfigurable systems”. In: *IEEE International Conference on Systems Engineering*. 1991, pp. 139–142.
- [129] Jonas Sticha. “Validating the Real-Time Capabilities of the ROS Communication Middleware”. BA thesis. Technische Universitt Mnchen, June 15, 2014.
- [130] Jan Stoess. “Towards effective user-controlled scheduling for microkernel-based systems”. In: *ACM SIGOPS Operating Systems Review* 41.4 (2007), pp. 59–68.
- [131] *Technical Safety Concept Status Report*. AUTOSAR, Oct. 13, 2010.
- [132] Sven Tomforde, Bernhard Sick, and Christian Mller-Schloer. “Organic Computing in the Spotlight”. In: *arXiv preprint arXiv:1701.08125* (2017).
- [133] Marcus Vlp, Adam Lackorzynski, and Hermann Hrtig. “On the Expressiveness of Fixed Priority Scheduling Contexts for Mixed Criticality Scheduling”. In: *Proc. WMC, RTSS* (2013), pp. 13–18.
- [134] *What Is Your Definition of Software Architecture*. Carnegie Mellon University: Software Engineering Institute, 2017.
- [135] Saman Taghavi Zargar, Vahid Salmani, and Mahmoud Naghibzadeh. “MMUF: An Optimized Scheduling Algorithm for Dynamically Reconfigurable Real-Time Systems”. In: *Information and Communication Technologies, 2006. ICTTA06*. 2nd. Vol. 2. IEEE, 2006, pp. 3486–3491.
- [136] Fengxiang Zhang and Alan Burns. “Analysis of Hierarchical EDF Pre-emptive Scheduling”. In: *IEEE*, Dec. 2007, pp. 423–434.
- [137] Fengxiang Zhang and Alan Burns. “Schedulability analysis for real-time systems with EDF scheduling”. In: *IEEE Transactions on Computers* 58.9 (2009), pp. 1250–1258.
- [138] Werner Zimmermann and Ralf Schmidgall. *Bussysteme in der Fahrzeugtechnik: Protokolle, Standards und Softwarearchitektur ; mit 103 Tabellen*. 5., aktualisierte und erw. Aufl. ATZ/MTZ-Fachbuch. OCLC: 890312433. Wiesbaden: Springer Vieweg, 2014. 507 pp.